一．Guest OS页表更新：

VMM对于客户机物理页信息的维护主要包括引用计数、所有者(虚拟机id)和页类型。对每个客户机物理负的所有者和用途进行跟踪，使得VMM能够对不同的虚拟机进行安全隔离。

客户机物理页按用途可以分为可写页、页表页、页目录页和描述符表页。这些页的类型是互斥的，也就是说，一张客户机物理页不可能同时为两种类型。并且对于每种类型，VMM都维护了一个独立的引用计数，在对页类型进行修改时，该页类型的引用计数必须为零。在Guest OS的页表映射中，页表页、页目录页和描述表页都必须是只读的。这保证了Guest OS不能随意修改自己的页表，也就是Guest OS创建、修改、删除页表的行为都会被Hypervisor所截获，从而保证了一个虚拟机不能随意访问其他虚拟机的内存。

在不能直接修改页表的情况下，Guest OS需要通过相关的超级调用(Hypercall)来更新页表，VMM会对相应页表项进行严格的检查和验证。在进行实际更新之前，VMM会对更新的请求进行验证，任何试图映射到所分配的内存之外物理页的更新都会被会禁止，任何创建到页表页的可写映射的更新也会被禁止。进行验证时，VMM所维护的页类型和引用计数就起关键作用。例如，Guest OS不能创建到页表页的可写映射，因为这要求该页既为页表页，又同时为可读写页，违反了类型的互斥原则。因为超级调用是开销比较大的操作，所以可以对页表的更新进行批处理，即在一次超调用中进行多个页表的更新。此外，Guest OS更新M2P映射表（位于xen）需要VMM的验证。

PV模式下的Xen对于虚拟地址转换采用的是直接模式。当新分配一个Page时，会首先为该Page分配页表项（PTE），然后PTE指向该Page。Hypervisor控制着所有Guest OS的物理地址和及其机器地址关系，通过P2M表（位于guest OS），Hypervisor获得该Page的机器地址。在Guest OS中，Page对应的PTE项指向物理地址，Hypervisor修改PTE项将物理地址修改为机器地址，这样Guest OS访问某一线性地址就能够直接通过CPU的MMU直接找到相应的机器地址。

二. 页表更新分析：

mmu\_context.h: activate\_mm; switch\_mm

很重要的结构体：mmu.c: static const struct pv\_mmu\_ops xen\_mmu\_ops 页表管理中涉及到的hook函数：pv\_mmu\_ops.dup\_mmap=xen\_dup\_mmap;

**页表的建立：**

考虑常用的fork()调用，当页表创建完成时，其control flow如下：

do\_fork->copy\_process->copy\_mm->dup\_mm->dup\_mmap->copy\_page\_range

为新进程建立新的页表映射，并将每一个pte页表，置为只读，以便激发起写时复制技术，从而完成写时复制的准备工作，接着需要设置相应页的页类型，并且绑定相应的页类型。

dup\_mmap->arch\_dup\_mmap->xen\_dup\_mmap->xen\_pgd\_pin->\_\_xen\_pgd\_pin

\_\_xen\_pgd\_pin->\_\_xen\_pgd\_walk->xen\_pin\_page-> MULTI\_update\_va\_mapping

该control flow确保各级页表为只读并绑定相应页表类型。

xen\_pgd\_pin->\_\_xen\_pgd\_pin->xen\_do\_pin->xen\_extend\_mmuext\_op -> HYPERVISOR\_mmuext\_op

在把整个页表的基址设置到cr3之前，该hypercall需要把该进程的根页表的页类型从可写页绑定成（pin）相应的页表类型。

**总结：**

**绑定页表类型原因：**

以32位支持PAE且页粒度为4KB的操作系统为例，guest OS 在为进程创建页表时，Hypervisor必须在将该套页表的基址加载到硬件寄存器cr3之前，首先验证该套页表的有效性，例如，不能允许Guest OS创建某些页表页的可写映射。这意味着Hypervisor需要对该套页表的页表项进行递归验证，因此造成大量的性能开销。

**改善方法：**为了避免每次上下文切换所带来的性能开销，Hypervisor引入了pin和unpin两种行为。具体来说，guest OS首先填好页目录指针表的表项，接着填好对应下一级的页目录表的表项，最后填充好下一级对应页表的表项，注意在这过程中，为了方便guest OS的行为，其所用的页类型均是可写的。整套页表建立完成后，再将所有相应的表项设置为只读，并将各级页表页设置为相应的页类型，这是通过hypercall: HYPERVISOR\_mmuext\_op(MMUEXT\_PIN\_LX\_TABLE, guest physical frame number of the (LX))

完成的，每级页表的类型绑定从低到高通过该hypercall完成。最后，就是加载cr3。cr3寄存器的加载是在进程调度的时候更新的，具体如下：schedule()->context\_switch()->switch\_mm()->load\_cr3(next->pgd)-> write\_cr3->xen\_write\_cr3

load\_cr3加载的是mm\_struct->pgd，即线性地址，而实际上加裁到cr3寄存器的是实际的物理地址，write\_cr3(\_\_pa(pgdir));在装载cr3寄存器时将线性地址通过\_\_pa转换成了物理地址了，所以cr3寄存器是装的是实实在在的物理地址。通过hypercall:HYPERVISOR\_mmuext\_op(MMUEXT\_NEW\_BASEPTR, guest physical frame number of the PGD (L3)) 设置好cr3。

这就是所谓的延迟绑定：late pin。

**页表项的更新：**

当一些页表项需要更新时，guest OS可使用hypercall: HYPERVISOR\_mmu\_update(MMU\_NORMAL\_PT\_UPDATE)。如果guest OS直接修改页表项，也会陷入到Hypervisor中。例如：fork()系统调用：在页表创建后，其可能的一个control flow如下：

dup\_mmap->copy\_page\_range->copy\_one\_pte->set\_pte\_at->xen\_set\_pte\_at (如何将页设置为只读，是Xen来做还是guest O来做，是透过哪个hypercall来完成???)

(调用HYPERVISOR\_mmu\_update，目的是把父子进程所共享的共享页设置为只读)。

**页表的销毁：???**

当该页表被销毁时，首先需要通过hypercall: HYPERVISOR\_mmuext\_op(MMUEXT\_UNPIN\_TABLE)可将相应各级页表的类型解除绑定，这样该套页表所在页就可以被guest OS所回收，而不再涉及Hypervisor，这就是初始解绑定（early unpin）。

**IOTLB刷新：**

前面讲到在建立页表时，HYPERVISOR\_mmuext\_op(MMUEXT\_PIN\_LX\_TABLE, guest physical frame number of the (LX))需要把整个页表的基址设置到cr3，也就是把该进程的每一级页表的页类型从可写页绑定成相应的页表类型。具体来说，Hypervisor主要做了两件事：1. 通过page table walk做了上述检查后，将每一级页表的物理页类型设置为页表类型；2. 并将每一级的页表页绑定到对应的页表页类型上。在这个过程中，我们注意到，IOTLB会发生刷新，这个是显而易见的。因为Hypervisor通过I/O页表建立起客户物理地址和机器地址间的映射，从而使外设只能访问客户机指定的内存区域，其页类型当然不会是页表类型，以避免DMA攻击。因此，当一个物理页其页类型由可写页被设置成页表时，Hypervisor必须把该物理页从I/O页表中清除，解除相应的映射关系，这是在iommu\_unmap\_page实现的。当解除了相应的页表项后，其IOTLB必须被刷新，以保证I/O页表的同步和安全。接下来我们详细分析Hypervisor在接收到相关请求时所要执行的函数流。

1. NEWBASEPTR在放到cr3之前，也要对每级页表进行验证
2. op.cmd=PIN\_L3\_TABLE 只对L3页表进行pin，其他页表的pin是在其他命令时进行，如：op.cmd=PIN\_L2\_TABLE

三．页表类型页合法的两个标志：1. 页类型被验证有效；2.页类型已被绑定。这一节我们具体分析Hypervisor是如何在do\_mmuext\_op 流程（op.cmd=PIN\_L3\_TABLE）函数里做到的。