x86虚拟化概述

系统软件 2016-11-19 00:45

本文发自 http://www.binss.me/blog/An-overview-of-the-virtualization-of-x86/，转载请注明出处。

最近CSP课程讲了x86的虚拟化，结合一些Paper和资料，总结成该笔记。如有理解错误之处，欢迎指出。

引子

提到虚拟化，我们经常会看到VMM和Hypervisor两个词，实际上它们说的是同一个东西(后文统一用VMM，因为比较短)，是用来创建和运行虚拟机的软件、固件或硬件(wiki)。在虚拟化架构中，它通常拥有最高的权限，负责资源管理、VM之间安全保障等工作，而VM需要运行在VMM之上。按hypervisor运行的层次不同，可以把它可以分为Type 1和Type 2两类，即Native VM和hosted VM。

Type 1在硬件和操作系统之间，添加了一个虚拟化层，不需要通过操作系统而直接访问硬件，相对于Type 2效率更高，且具有更好的可扩展性、健壮性和性能。常见的有VMware ESX / ESXi、Microsoft Hyper-V等。 Type 2将虚拟化层直接安装到干净的x86系统上，适应性更好，目前大多数普通用户使用的都是Type2的虚拟机，如VMware Wrokstation / Fusion、VirtualBox、Parallels Desktop等。

(但对于KVM或bhyve这类kernel module虚拟机来说似乎无法说清楚它们是Type 1还是Type 2，一方面，它们的host做了VMM的工作，另一方面，它们依然依赖于host OS)

本文主要讨论x86下各硬件的针对虚拟化的解决方案，因此下文不会对Type进行区分。

CPU虚拟化

x86分为了ring0-ring3(指环王的即视感)共4个权限级。操作系统是处于最高级别的ring 0，应用程序处于最低级别的ring 3。在这样的架构下，操作系统被设计成直接运行在硬件上，拥有硬件的全部控制权。

在虚拟化架构中，VMM(hypervisor)作为GOD，掌管虚拟机的创建和管理，自然必须运行在ring 0。而为了隔离，Guest OS只能退而求其次，运行在ring 1(Privilege deprivileging)。由于不是运行在ring 0，对于特权指令，Guest OS在运行时会引发一个异常(trap)，从而被VMM捕获。VMM在检查通过后将代为执行，然后将结果返回给Guest OS。而对于一部分敏感指令(17条)，Guest OS的执行权限不够，本该交给VMM来执行。然而它们在非ring 0下可以依然照样执行，只是和ring 0下执行的语义会有不同，或者直接静默失败了。这个锅在于x86在设计时没有考虑到虚拟化的需求，不满足"敏感指令都是特权指令"这个虚拟化定律。如何解决这个问题，成为了CPU虚拟化的关键。

全虚拟化

对于用户级的指令，可以直接送到CPU执行。对于对于特权指令，需要通过陷入模拟(Trap-and-emulation)到VMM让其代为执行。而对于那17条敏感指令则需要通过二进制翻译系统解决。

二进制翻译系统是位于应用程序和硬件之间的一个软件层，可以将某种特定的二进制代码翻译为另一种架构下的二进制代码。在虚拟化中的二进制翻译可以对这17条敏感指令的代码进行替换，从而避免了不确定行为的发生。同时，为了提高效率，二进制翻译后的代码会进行缓存，避免每次都进行翻译。

代表作：VMware workstation、VMware ESX Server。

优点：无需对Guest OS进行修改，兼容性强。

缺点：需要执行大量的二进制翻译工作，性能成为瓶颈。

半虚拟化

半虚拟化从另外一个角度解决了这个问题：与其千方百计去捕获这17条指令，直接不用这些指令不就好了吗？于是就得对Guest OS进行一些修改，替换掉这些指令，转而调用VMM提供的特殊API(hypercall)来进行模拟。当Guest OS需要执行敏感操作时，直接通过hypercall调用VMM，避免了捕获的开销。

代表作：Xen

优点：比起全虚拟化，由于避免了trap和翻译，性能自然大大提升。

缺点：需要对Guest OS的系统内核的深度修改，使其能够通过调用hypercall来执行敏感指令，面对各种系统，工作量非常大。由于Windows不开源，没办法进行修改，因此不支持Windows。同时，Guest OS将与VMM强耦合，增加了系统维护成本，降低了可移植性。

硬件辅助虚拟化

这时候业界大哥Intel站出来了，说大家灌Paper灌得那么幸苦，今后不用忙活了，你们说x86对虚拟化支持不好，那我改就是了嘛。于是扔出了VT-x。当然AMD也扔出了AMD-V。两者都针对特权指令为CPU添加了一个工作模式，分为VMX Root Operation mode和VMX Non-Root Operation mode。VMM跑在root mode下，Guest OS跑在non-root mode level的Ring 0上，应用跑在Ring 3。

因此在硬件辅助的完全虚拟化中，VMM不需要费尽心思去捕获特权指令了，CPU会捕获non-root mode的Ring0上执行的特权指令，然后VM-Exit到root mode的VMM进行处理，处理完后通过VM-Entry返回。从硬件层面解决了捕获的问题。

此外还引进了VMCS来保存vCPU的寄存器状态，使得在VM-Exit和VM-Entry时不必将寄存器的值读/写到内存。

代表作：VMWare Fusion、Parallels Desktop(实测发现如果不开VT-X的话虚拟机会慢出翔)

优点：实现了高性能的全虚拟化。

缺点：需要硬件支持(如今的CPU基本上已支持)。由于还是通过trap解决，导致性能上可能还比不上半虚拟化。

内存虚拟化

x86使用了虚拟地址-物理地址的内存映射。每个进程都拥有一个虚拟地址空间，然后由MMU负责地址的转换。

MMU(内存管理单元)是x86 CPU中负责地址转换的模块，当CPU在把发送地址(VA)时，MMU会截获该地址，将该地址转换为物理地址(PA)然后才发给内存。地址转换以page(4K)为单位，通过页表(page table)来实现，页表中的每一项都记录了VA到PA的映射关系。由于页表经常会被用到，但却是存放在内存中的，因此MMU内置了TLB(Translation Lookaside Buffers)来存放最常使用的页表项。每次转换时MMU首先会到TLB中去找，找不到才到内存的页表中找，并按照一定策略将最常用的页表项换到TLB中。MMU还提供内存访问权限检查。根据PTE条目中对访问权限的限定检查该条VA指令是否符合，若不符合则不继续，并抛出异常。

在虚拟化环境中，内存要被多个Guest OS使用，而此时Guest OS上的地址空间再也不能简单的对应到硬件的地址空间上了(不然别的OS怎么办)。怎么办呢?

Wheeler有一句名言，All problems in computer science can be solved by another level of indirection，那么我们再引入一层内存映射不就好了嘛?

于是全虚拟化引入了一层Guest OS的物理地址(PA)到宿主机机器地址(MA)的映射。因此我们有两个映射，VA到PA，PA到MA，这样就为Guest OS模拟了一个连续、平坦的地址空间，符合了Guest OS的需求。在这样的基础上，依然诞生了全虚拟化、半虚拟化和硬件辅助虚拟化三种内存虚拟化方式。

全虚拟化

每次对内存的访问都要先从VA到PA，再从PA到MA，但这和已有的架构不兼容(x86的CR3、MMU都是为一次地址转换设计的)。于是有人提出直接维护VA到MA的映射，即影子页表(Shadow Page Table, SPT)。但是VA到PA的映射是在不断变化的，如何能够让影子页表保持最新的映射呢？解决方法非常巧妙，将VA到PA的页表设为只读，那么每次Guest OS要改页表时会触发page fault，trap到VMM，然后VMM就知道要去更新影子页表了，于是影子页表根据新的VA到PA的映射，修改VA到MA的映射。这样在进行内存访问时只需进行一次地址转换，故MMU可以通过影子页表进行寻址。此时CR3只须指向SPT即可。

代表作基本上还是CPU虚拟化的那几个，后文将不再提及。

优点：巧妙地实现了虚拟机内存映射。

缺点：每个进程有一张页表，假设有10台VM，每台VM上运行100个进程，那么就需要维护10 * 100 = 1000个影子页表。这耗费了大量的空间，同时更新影子页表的开销也不容小视。实现起来也非常复杂。

半虚拟化

Xen说了，为什么一定要有两层映射，如果Guest OS都老老实实地使用VMM给它分配的那块地址空间，那么Guest OS、host和VMM就能够使用同一片内存地址空间了。但问题依旧，Guest OS不知道正和别人共享着同一片地址空间，会觉得整片空间都是它的。怎么办呢，老办法，改Guest OS，让其知道自己运行在虚拟化环境中，从而直接去访问“不连续”的MA，这种方法被称为direct paging。

在这种情况下，PA到MA的页表(P2M)保存在Guest OS中，Guest OS进行内存访问时能够自己完成VA-PA-MA的转换，然后MMU通过VA-MA进行寻址。为了保证安全性，将设为只读，对页表进行修改需要通过hypercall调用VMM来进行。

还有一个只读的M2P，弄不明白实际用在什么场合。

硬件辅助虚拟化

这个时候Intel又跳出来了，说以往需要这么折腾，不就是因为MMU不支持两级转换嘛，现在给你们搞一个。于是提出了EPT(Intel Extended Page Table)。

EPT引入了EPT和EPT base pointer，EPT中存储着PA到MA的映射，而EPT base pointer负责指向EPT页表。当进行地址转换时，根据CR3找到VA-PA的页表进行VA到PA的转换，然后根据EPT base pointer找到EPT，进行PA到MA的转换。

AMD提出的是NPT(Nested Page Table)，此时Guest OS和Host都有自己的CR3。当进行地址转换时，根据gCR3找到VA-PA的页表进行VA到PA的转换，然后根据nCR3找到PA-MA的页表，进行PA到MA的转换。

两种技术大同小异，都通过硬件手段对两次地址转换提供了支持。这种方式又称为nested paging。

优点：无需维护影子页表，解决了内存全虚拟化的缺点。

缺点：两级页表查询，假设有4层页表映射(页目录表-页表-页表项-地址)，则需在每层都查询两级页表，带来巨大开销，只能寄望于TLB命中。

I/O虚拟化

I/O设备之间也需要进行虚拟化，以隔离和高效为目的。

全虚拟化

在全虚拟化中，通过对设备进行模拟来实现虚拟化。VMM会在Guest OS上模拟出一个虚拟硬件设备供其使用，比如VMware的8139网卡。对于Guest OS来说，它并不知道设备是虚拟出来的，于是会通过自带的设备驱动去访问VMM模拟的IO设备，对于端口I/O，由于是特权指令，会trap到VMM中，交给设备模拟器进行模拟；对于MMIO，VMM把映射到该MMIO的页表设为无效，访问时会触发page fault并trap到VMM中，交给设备模拟器进行模拟；对于中断，设备模拟器在接收到物理中断并需要触发中断时，发送一个虚拟中断给Guest OS。

优点：兼容性好，可以模拟出真实不存在的设备。

缺点：I/O访问伴随着大量的trap，并且寄存器由软件模拟，性能低。

半虚拟化

通过分离驱动模型实现。

通过修改Guest OS，为其安装特定的驱动，称为frontend，以抽象设备接口(如SATA/IDE)的形式暴露给Guest OS。当用户发出I/O请求时，会通过frontend转发到VMM/Domain 0中的backend，由其负责将请求发送到真正的物理设备驱动上。在完成请求后，沿原路将结果传递给Guest OS。

在Xen中，通过share memory + I/O buffer ring(放请求和响应，带有share memory的位置) + event channel(通知) + grant table(授权) 实现。

优点：比起全虚拟化对设备进行模拟，性能大大提升。

缺点：需要对操作系统进行修改，使其支持frontend驱动。

硬件辅助虚拟化

之所以要折腾I/O虚拟化，无非是硬件不够用的问题，非常好办，接入多个IO设备不就好了吗？比如你有几台VM，我就插几张网卡，保证每台VM都能分到一张物理网卡，该方法名为Direct I/O。然而即使这样，依然需要处理设备之间的DMA和中断隔离，需要有一种方案来实现DMA隔离(DMA Remapping)和分离不同设备的中断与路由(Interrupt remapping)。

这时Intel提出了VT-d技术，在北桥中内置提供DMA虚拟化和IRQ虚拟化硬件，通过硬件层的映射使得虚拟机内的IO请求直接映射到实际硬件上，让VM以独占的方式直接使用设备。

用BDF的Bus找到根条目，从其中的CTP找到上下文条目表，用BDF的Device+Function找到上下文条目，从其中的ASR找到设备对应的I/O页表。然后DMA重映射硬件利用该页表进行地址转换，将PA转为MA，从而使设备能够直接访问相应的内存。

AMD也提出泪湿的解决方案，称为IOMMU，它为每个设备分配一个保护域，规定了对I/O页的访问权限，只有拥有权限的设备才能访问到相应的内存。

所有的DMA请求都会被截获，可以对DMA中的地址进行转换使设备只能访问到规定的内存。

优点：解决了全虚拟化I/O中的性能问题，获得了接近原生的性能。

缺点：购买多个I/O设备开销较大，同时主板上的插槽有限，不具可扩展性。另外对于每一个I/O设备可能资源利用率不高。

为了解决Direct I/O的可扩展性问题，PCI-SIG组织(Intel是创始者之一)跳出来了，发布了SR-IOV(单根虚拟化)。支持SR-IOV的PCIe设备能够在硬件层直接虚拟化成多个虚拟设备(VF)供VM使用，然后通过VT-d的那一套实现VM对VF的直接访问。

虚拟出多个PCIe网卡，每个网卡具备独立的I/O功能。结合VT-d或IOMMU，可以实现VM直接访问VF。

优点：接近原生的性能，同时保持了较好的扩展性。

缺点：需要购买支持SR-IOV的硬件。

显卡虚拟化

不了解。待日后补充。

总结

全虚拟化采用模拟的方式来实现虚拟化，努力提供一个native的环境供虚拟机运行，无需Guest OS进行修改。

半虚拟化采用修改Guest OS的方式来实现虚拟化，让Guest OS知道自己运行在虚拟环境下，然后与VMM协同工作。

可以发现这是完全不同的设计哲学。

而硬件辅助虚拟化通过硬件解决了全虚拟化中的大部分问题。使得全虚拟化的性能大大提升，于是越来越成为主流。

剩下的是我自己的一些纠结：

曾经我以为全虚拟化就是不对Guest OS进行修改，Guest OS无法意识到自己运行在虚拟化平台上。当我看《Memory resource management in VMware ESX server》时，由于ESX属于全虚拟化，于是我也想当然地认为里面的balloon module是系统自带的了(因为之前好像看过Linux 3.x后集成了balloon之类的)，结果被打脸了，balloon driver是另外装的。这意味着Guest OS只需检查一下自己身上有没装balloon驱动就可以发现自己是否运行在虚拟机中。于是只能修改自己的理解：需要安装驱动并不意味着是不是全虚拟化，不需要修改内核就可以认为是全虚拟化。

参考

《Xen and the Art of Virtualization》

《Memory resource management in VMware ESX server》

《虚拟化技术原理与实现》

https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/VMware_paravirtualization.pdf

https://wiki.xen.org/wiki/X86_Paravirtualised_Memory_Management

https://my.oschina.net/jerikc/blog/228869