12 | 内存虚拟化：云原生时代的奠基者

2021-11-19 海纳

《编程高手必学的内存知识》 课程介绍 >

讲述：海纳
时长 23:33 大小 21.58M

你好，我是海纳。

今天的这节课呢，是软件篇中的最后一节课了，在前面的课程里，我们整体介绍了单机系统上内存管理的基础知识。这节课，我们就结合前面学习的内容，一起来探讨下，虚拟化中的内存管理，因为我们前面讲过了内存知识，在这个基础上，你再来学习虚拟化中的内存管理，就会简单多了。

当前，云计算已经成为各种网络服务的主流形式，但云计算不是一蹴而就的，它的发展也经过了长期的探索和演变。在演变的过程中，扮演核心角色的就是主机虚拟化技术。它历了虚拟机和容器两大阶段，其中**虚拟机以 VMWare 和 KVM 等为代表，容器以 Docker 为代表**。

虽然现在 Docker 技术非常火爆，甚至某种程度上，人们在讨论云化的时候往往就是指容器化，但是虚拟机技术在长期的发展中，也留下了非常宝贵的技术积累，这些积累在各种特定的场景里还在发挥着重要作用。

举一个我曾经遇到过的一个真实案例：在 Windows 上快速预览 Android 游戏。这个操作听起来很神奇是吧，那如何才能打破架构上的壁垒，达到快速执行的目的呢？这就需要对虚拟化的基本原理掌握得比较好，从而在虚拟机层面做很多优化。所以掌握虚拟化技术绝不仅仅只应用于云服务的场景，它可能会在各种意想不到的场景中发挥奇效。

这节课，我将带你从内存入手来学习虚拟化技术，当然，在学习内存虚拟化之前，我们还是需要先了解一下虚拟化技术中的一些关键技术点，以及几个核心的角色，这些背景是贯穿虚拟化技术的核心所在。

虚拟化技术的基本概念和原则

我们先来进行一下名词解释，**在虚拟化技术中涉及的有三个核心角色，分别是宿主机，客户机和虚拟机监控器**。宿主机，也被称为 Host，一般指代物理主机。客户机，也被称为 Guest，是指运行在宿主机上的虚拟机。而负责为客户机准备虚拟 CPU，虚拟内存等虚拟资源，并同时对客户机进行管理的模块，就是虚拟机监控器 (Virtual Machine Monitor，VMM)。

对于传统的物理主机而言，往往有很强大的计算资源，所以通常一台机器上会有多个用户共同使用，这样才能使得主机资源得到充分利用。但是在传统的单机系统里边，多个用户之间往往会相互影响。例如，通过 ps 命令可以看到系统上其他用户启动的进程，也可以使用 kill 命令杀死其他用户的进程，这就对隐私性和安全性带来了比较大的挑战。

针对上面的问题，**虚拟化技术可以让用户相互隔离开**。在不同的虚拟机实例中运行的用户，虽然运行在同一个物理主机上，但是相互无法看到对方，这样就很好的保证了虚拟机用户的隐私与安全。**在所有的虚拟化实现方案中，内置于 Linux 内核的虚拟化技术，也就是基于内核的虚拟机 (Kernel based Virtual Machine, KVM) 是影响力比较大的一个**。这也是我们这节课的重点。

第三个比较重要的组件是 VMM，有些资料中也把它叫做 Hypervisor。正如它的名字的含义，它是负责管理和调度虚拟机的，虚拟机在执行特权指令、处理中断和管理内存等特殊

操作时，都需要通过 VMM 来完成相应功能。了解完虚拟化技术的三个核心角色后，接下来，我来介绍下虚拟化技术在实际工作中需要遵循的指导原则。

1974 年，两位计算机科学家 Gerald Popek 和 Robert Goldberg 发表了一篇重要的论文《虚拟化第三代体系结构的正式要求》，在这篇论文中提出了虚拟化的三个基本条件：

1. **等价性**，即要求在虚拟机环境中运行的程序，应当与在物理机上运行的程序行为一致，且所有能在物理机上运行的程序都应该能够在虚拟机中运行；
2. **资源限制**，即要求虚拟机使用的资源需要被进行监督和限制，虚拟机不能越界使用到不属于它的资源；
3. **高效性**，即要求在虚拟机中运行的程序与在物理机中运行的程序相比，性能应该无明显的损耗。

这三个条件便为后续的虚拟化技术的发展提供了有效的指导原则，设计良好的虚拟化技术需要同时满足以上三个条件。

第一个等价性的条件自然不需要过多解释，如果 Guest 里运行的程序结果都不能保证，那么虚拟化的技术就没有任何意义了。因此，**等价性是虚拟化技术中最基本的要求**。

至于第二个资源限制的条件也比较容易理解，**我们使用 Guest 的目的本身就是为了对资源使用进行限制与管理，防止不同用户之间对计算机资源的相互干扰，这也是我们考虑使用虚拟化技术时最重要的原因**。

在前两个条件的约束下，我们可以很容易想到一个实现虚拟化技术的方案是：**通过纯软件模拟 CPU 执行过程**。也就是说，这里需要对完整的底层硬件进行模拟，包括处理器、物理内存和外部设备等等。这样的话，Guest 的所有程序都相当于运行在 Host 的一个解释器里，来一条指令就解释一条指令，资源限制以及运行等价的要求都很容易满足。

不过这个方案的缺陷也非常明显，就是无法满足三个条件里面的高效性。因为你是用软件来对 CPU 的指令进行了翻译，通常一条指令最终会被翻译成非常多的指令，那效率自然也是非常低的。既然对指令进行翻译的效率是如此低下，那我们为什么不能让 Guest 程序的代码直接运行在 Host 的 CPU 上呢？

我们本来翻译指令的目的，是为了让 VMM 能够对 Guest 执行的指令进行监管，防止 Guest 对计算资源的滥用，那如果又让 Guest 的执行直接运行在 CPU 上，VMM 又哪里有机会能够对 Guest 进行监管呢？

为了解决这个问题。人们提出一个重要的模型，这就是**陷入模拟（Trap-and-Emulate）模型**。接下来，我们就来了解一下吧。

Trap-and-Emulate 模型

陷入模型的核心思想是：**将 Guest 运行的指令进行分类，一类是安全的指令，也就是说这些指令可以让 Host 的 CPU 正常执行而不会产生任何副作用，例如普通的数学运算或者逻辑运算，或者普通的控制流跳转指令等；另一类则是一些“不安全”的指令，又称为“Trap”指令，也就是说，这些指令需要经过 VMM 进行模拟执行，例如中断、IO 等特权指令等。**

接下来，我们来看一下它的具体实现过程：对于“安全”的指令，Guest 在执行时可以交由 Host 的 CPU 正常运行，这样可以保证大部分场景的性能。不过，当 Guest 执行一些特权指令时就需要发出 Trap，通知 VMM 来接管 Guest 的控制流。VMM 会对特权指令进行模拟 (Emulate)，从而达到资源控制的效果。当然在进行模拟的过程中需要保证执行结果的等价性。

经过这样一个 Trap-and-Emulate 的过程，Guest 就可以在保障等价性以及资源限制的前提下，尽可能地满足虚拟化的高效性的条件。

可能你对此感知不深，下面我给你举一个例子，你就能理解 Trap-and-Emulate 到底是怎么回事了。

以 0x80 号中断为例，在第 2 节课里，我们使用 0x80 号中断，调用了 write 这个系统调用，在控制台上打印文字。"int 0x80"这条指令就是一个特权指令，它会导致当前进程切入内核态执行。在虚拟化场景下遇到这种特权指令，我们不能直接交给宿主机的真实 CPU 去执行，因为宿主机 CPU 会使用宿主机的 IDT 来处理这次中断请求。

而我们真正希望的是，**使用客户机的 IDT 去查找相应的中断服务程序。**这就需要 Guest 退回到 VMM，让 VMM 模拟 CPU 的动作去解析 IDT 中的中断描述符，找到 Guest 的中断

服务程序并调用它。在这个例子中，Geust 退回 VMM 的操作就是 Trap，VMM 模拟 CPU 的动作去调用 Guest 的中断服务程序就是 Emulate。

现在，我们有了整体的方案，不过，这里仍然存在一个问题：当 Guest 的内核代码在 Host 的 CPU 上执行的时候，Guest 没有办法区分“安全”指令和“非安全”指令，也就是说 Guest 不知道哪条指令应该触发 Trap。幸好，现代的芯片对这种情况做了硬件上的支持。

现代的 X86 芯片提供了 VMX 指令来支持虚拟化，并且在 CPU 的执行模式上提供了两种模式：**root mode 和 non-root mode，这两种模式都支持 ring 0 ~ ring 3 三种特权级别**。VMM 会运行在 root mode 下，而 Guest 操作系统则运行在 non-root mode 下。所以，对于 Guest 的系统来讲，它也和物理机一样，可以让 kernel 运行在 ring 0 的内核态，让用户程序运行在 ring 3 的用户态， 只不过整个 Guest 都是运行在 non-root 模式下。

有了 VMX 硬件的支持，Trap-and-Emulate 就很好实现了。Guest 可以在 root 模式下正常执行指令，就如同在执行物理机的指令一样。当遇到“不安全”指令时，例如 I/O 或者中断等操作，就会触发 CPU 的 trap 动作，使得 CPU 从 non-root 模式退出到 root 模式，之后便交由 VMM 进行接管，负责对 Guest 请求的敏感指令进行模拟执行。这个过程称为 **VM Exit**。

而处于 root 模式下的 VMM，在一开始准备好 Guest 的相关环境，准备进入 Guest 时，或者在 VM Exit 之后执行完 Trap 指令的模拟准备，再次进入 Guest 的时候，可以继续通过 VMX 提供的相关指令 VMLAUNCH 以及 VMResume，来切换到 non-root 模式中由 Guest 继续执行。 这个过程也被称为 **VM Entry**。

在理解了 VMM 的基本工作原理以后，我们就可以探讨虚拟化场景下的内存管理了。接下来，我们从虚拟机用户的视角出发，来看看 VMM 是如何支持 Guest 的内存管理的。

Guest 内存管理

既然讲到内存管理，我们先来研究下物理机和虚拟机分别是怎么获取系统的内存信息的。

在 x86 架构的实模式下，系统启动时，BIOS ROM 会被映射到内存的 0xF0000 ~ 0xFFFFF 的位置。CPU 上电后会从 0xFFFF0 的位置开始执行，这里会跳转到 BIOS 的起始

代码中。BIOS 的代码会检查物理内存的信息，并记录下来。

之后，操作系统可以通过查询 INT 15h 的中断，来获取物理内存的信息，然后根据寄存器 AX 值的不同来返回不同的内存信息。当 AX 值设置为 0xE820 时，将返回所有已安装 RAM 以及 BIOS 保留的物理内存范围的内存映射。我们看到，在物理机下，是通过 INT 15h 这个中断来获取系统的内存信息的。

类比下来，如果虚拟机里边想要获取系统的内存信息的话，就需要 VMM 模拟物理机 BIOS 的行为。在系统启动时，VMM 会将模拟 BIOS 的代码直接放到内存的 0xF0000 ~ 0xFFFFF 的位置。在构建中断向量表的时候，则将第 0x15 位置的中断函数地址设置为虚拟的内存查询函数地址。而调用 INT 15h 的中断时，中断服务程序返回的是用户配置的 Guest 的内存信息。这样的话，就可以使得 Guest 以为自己已经获取了实际物理机的内存信息。

我们知道，**在 x86 的架构上，系统启动时需要先在实模式下完成系统的引导，然后才会进入保护模式**。同样，**Guest 在启动过程中也需要先通过实模式进行引导，再切换到保护模式下**。所以，我们学习 Guest 的访存机制也是需要分别考虑实模式跟保护模式下的不同处理方式。

实模式 Guest 的访存

正常情况下，当一个 Host 系统中启动运行 Guest 系统时，此时的 Host 是处于保护模式的，而 Guest 则因为刚启动，所以需要运行在实模式下。此时又碰到一个问题，Guest 里实模式的代码又如何运行在 Host 处于保护模式下的 CPU 上呢？

这个问题同样需要硬件来支持。在 x86 体系的 CPU 中，可以支持一种虚拟 8086 的模式，这个模式又被称为虚拟 - 实模式，意思是可以让 CPU 在保护模式下来运行实模式的程序。当然这里虚拟 8086 模式下访问的地址，并不意味着程序跟实模式一样，就可以直接访问 Host 的真实物理地址了，只是说在该模式下，程序可以采用同实模式下一样的寻址方式，但访问的地址还是 Host 的虚拟地址，但在 Guest 自己看来，它认为自己访问的是 Guest 的物理地址 (Guest Physical Address,GPA)。

这种情况下，Guest 代码中的逻辑地址到 Host 的物理地址（Host Physical Address, HPA）的转换主要分为三个步骤：

1. 我们知道实模式下程序访存时是通过段式寻址的方式，也就是说，Guest 程序的逻辑地址可以通过分段机制转换为 GPA，这个过程是由 Guest 模式下 CPU 自发地进行，需要 CPU 运行在上边提到的虚拟 8086 模式下；
2. Guest 的物理内存可以由 VMM 转换到 Host 的虚拟内存地址（Host Virtual Address，HVA）。这一步的转换过程可以由 VMM 内部维护的数据结构进行查表得出；
3. 最后一步的转换也是由 VMM 直接调用缺页中断服务函数 (get_user_pages) 将 Host 的虚拟内存映射到物理页。你要注意的是，这一步是 VMM 主动调用的，而不是由中断触发的。

我们知道，在物理机上进行虚拟地址与物理地址转换的话，需要 cr3 寄存器来存放页表。因此，在 Guest 的实模式下，为了能够获取到实际运行的物理地址，我们需要在 VM Enter 的过程中将 cr3 寄存器设置成 VMM 为 Guest 准备的页表。

在实模式下，因为 Guest 指令访问都是物理地址，所以 cr3 寄存器还需要放置负责映射 GPA 到 HPA 的页表基址。在初始状态下，VMM 只需要准备一个根页面就可以了，等运行到缺页异常时，再通过缺页异常处理函数，来按需要完成页面的映射。

保护模式下 Guest 的访存

我们说，运行在实模式的 Guest 只需要一个页表就可以完成 GPA 到 HPA 的映射。但在保护模式下，我们知道每个进程都有自己的页表，维护着 GVA 到 GPA 的映射。所以保护模式下的内存转换方式要更加复杂。

在保护模式下的进程，当 Guest 准备访存时，cr3 寄存器此时存放的是 Guest 的页表。如果将这个页表交给 MMU 去查询，得到的将是 GPA 的地址，而不是真正的 HPA 地址。这是因为从 GVA 到 HPA 之间存在三层映射关系，即：

GVA 到 GPA 的映射；

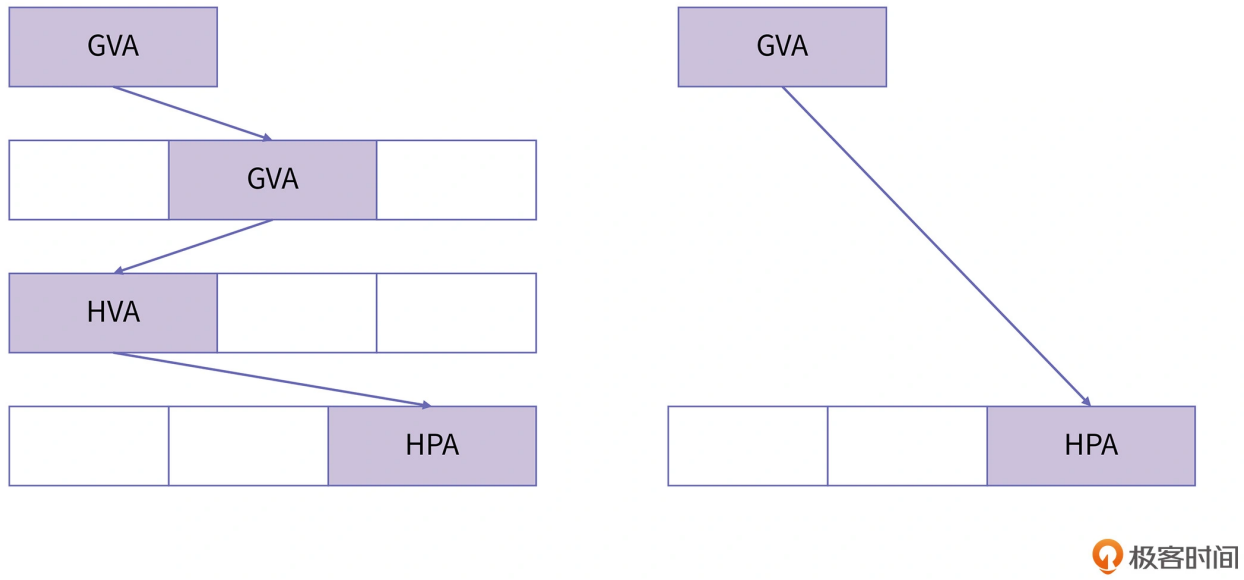
GPA 到 HVA 的映射；

HVA 到 HPA 的映射。

但 MMU 却只有一个。因此，要解决这个问题，我们需要将 cr3 寄存器中指向的 Guest 的页表，替换成为一张从 GVA 到 HPA 映射的页表。当 Guest 再进行访存时，则可以通过这

个页表完成完成从 GVA 到 HPA 的转换过程。因为在这个过程中，**新建的这张页表实际上会把 Guest 本身的页表给遮挡起来，所以我们称这个页表为影子页表 (Shadow page table)**。

我们说过，保护模式下每个进程都需要有自己的页表，同样的，VMM 也需要为 Guest 的每个进程维护一个影子页表。在 Guest 的进程切换过程，要更新 cr3 寄存器指向的页表地址，VMM 要把这个操作拦截下来，将 Guest 页表换成影子页表。影子页表的示意图如下所示：



可以看到，影子页表将原来的三级映射压缩成了一级映射，CR3 寄存器里只要存储影子页表的地址就可以了。这样 MMU 就可以自动完成从 GVA 到 HPA 的转换。

明白了影子页表的作用后，我们接下来看下在 KVM 里边影子页表是如何实现的。从影子页表的机制中我们可以看出，实现影子页表的过程中有两个关键点：

1. **cr3 寄存器的切换；**
2. **影子页表的构建。**

在第一点中，由于进程切换的时候都需要进行页表的切换，也就是对 cr3 寄存器的修改。因此，**当 Guest 在进程切换准备把 Guest 的页表写入 cr3 寄存器时，需要 VMM 介入进**

来，记录下此时要写入的 Guest 的页表，同时把 GVA 到 HPA 映射的影子页表写入到 cr3 中，完成一次偷梁换柱。

在第二点中，影子页表的构建，主要是通过影子页表的缺页异常处理函数来完成的，它主要的流程是：**当 Guest 执行访存指令，来进行访存的时候，会将 GVA 发送给 MMU 进行查找。由于此时 cr3 存放的是影子页表，因此 MMU 会通过影子页表来查找 GVA 对应的 HPA。如果找到了，就可以直接从 HPA 中读取对应的数据，然后流程结束。**

如果此时影子页表中还没有 GVA 到 HPA 的映射，就会触发 VM Exit，并从 Guest 模式退出到 Host 模式，由影子页表的缺页处理函数进行处理。影子页表的缺页处理函数会通过上文保存的 Guest 的页表，来查找 GVA 对应的 GPA。

如果 Guest 的页表中，GVA 到 GPA 的映射还不存在，就会由 VMM 向 Guest 注入缺页异常，并交由 Guest 的缺页异常处理函数，完成 GVA 到 GPA 的映射过程。完成映射后，Guest 会继续进行访存，由于此时影子页表中 GVA 到 HPA 的映射还未完成，CPU 此时会继续进入影子页表的缺页异常处理函数中。

当 GVA 与 GPA 的映射已存在时，就只需要根据 VMM 所维护的映射关系计算出 HVA。然后可以借助 Host 的内存管理机制，来分配空闲的物理页面，并且完成 GVA 到 HPA 的映射（与实模式相同，这一步也是由 VMM 主动调用 get_user_pages 完成的）。最后将映射关系填充到影子页表中。这就完成了影子页表的构建。

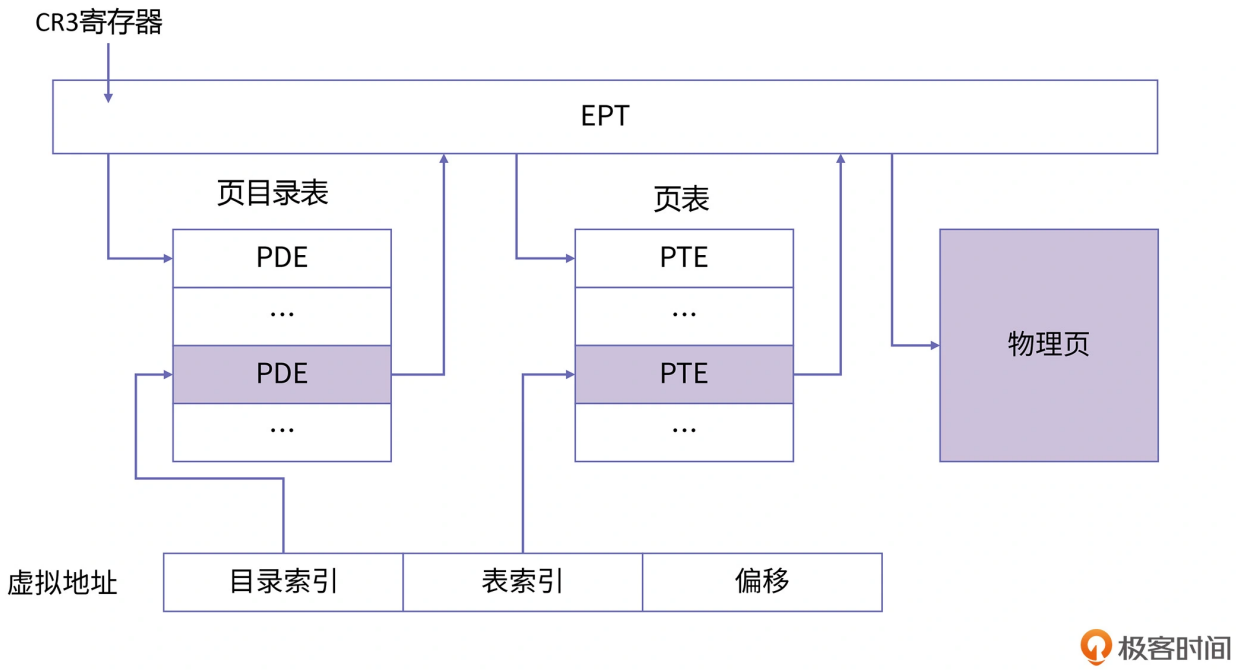
影子页表机制实际是一种纯软件实现的机制，我们可以看出，**影子页表是通过软件方式实现了 MMU 的能力**。而且在影子页表的使用过程中，会多次发生 VM Entry 和 VM Exit，你可以想象的到，影子页表机制的效率非常慢。为了解决这个问题，**硬件厂商通过新增一个页表转换单元来提升性能，这就是扩展页表 (Extended Page Table, EPT)**。

硬件支持——EPT

芯片厂商们为了提高虚拟化的效率，从硬件实现上支持了 2 层地址的翻译，其中 AMD 提出了嵌套页表（Nested Page Table，NPT）的机制，Intel 提出的是扩展页表（Extended Page Table，EPT）机制。其实，这两者的实现原理类似，只是命名有所不同。因为 Intel 的服务器芯片更加常见，所以我们这节课主要还是以 Intel 的 EPT 为例来进行讲解。

在增加 EPT 机制后，相当于有两个地址转换器，其中，MMU 负责 GVA 到 GPA 的地址转换，而 EPT 则负责 GPA 到 HPA 的地址转换。两个转换器在硬件上相互配合，MMU 根据 Guest 的页表翻译好 GPA 并传递给 EPT，EPT 通过 EPT 的页表翻译找到 HPA 的地址进行访存。同时 Intel 为了处理 EPT 的缺页，也引入了 EPT 的缺页异常机制，**它的原理与 MMU 原理一致**。

EPT 的工作原理如下图所示：



上面这幅图与第 2 节课中的图非常相似，不同之处在 CR3，PDE， PTE 中记录都是 GPA，而所有的物理地址都要经过 EPT 的翻译才能找到真正的 HPA。

对 Guest 页表来说，因为每个进程需要一个 Guest 页表，所以会维护多个 Guest 的页表。但对于 EPT 的页表来说，其本质是 GPA 到 HPA 的映射页表，因此一个虚拟机只需要维护一个 EPT 的页表就可以了。

硬件翻译的整个过程对与 Guest 来讲则是透明的，因此在 Guest 发生缺页异常则不需要再进行 Guest 与 Host 模式之间的切换了，也就节省了大量的上下文切换的开销。

总结

这一节课，我们一起学习了虚拟机是如何处理内存的。在这节课的开始部分，我们一起了解了虚拟化技术产生的动机和它的基本原理。

在这个基础上，我们又研究了虚拟机内部是如何管理内存的。在虚拟机 Guest 启动的时候，宿主机 Host 肯定是运行在保护模式的，也就是说，分页机制已经开启。但是 Guest 仍然运行在实模式下，所以 Guest 会采用虚拟 8086 模式运行。在这种情况下，因为 Guest 是直接操作 GPA 的，所以 VMM 只需要做好 GPA 到 HPA 的转换就行了。

当 Guest 进入保护模式后，Guest 也会维护自己的页表，我们把这个页表叫做虚拟机页表，也就是 gPT。显然 gPT 是不能将 Guest 的虚拟地址转换成真正的物理地址的，这就需要 VMM 来做一次处理，将 gPT 替换为自己精心准备过的页表，也就是影子页表，sPT。

不过，影子页表的维护和切换的效率十分低下，为了解决这个问题，硬件厂商提供了 EPT，它可以协助 MMU 进行第二次页表转换。也就是说，MMU 负责将 GVA 转换为 GPA，然后 EPT 再将 GPA 转换为 HPA，来完成真正的内存页映射。

学习完今天这节课，我相信你已经足够掌握内存虚拟化的大部分知识了。但是虚拟化技术中还有 CPU 虚拟化、中断虚拟化等等广泛的话题，如果你对虚拟化技术十分感兴趣的话，可以参考《KVM 虚拟化技术：实战与原理解析》和《深度探索 Linux 系统虚拟化：原理与实现》等书，这样你就可以对虚拟化技术有更全面的掌握了。

思考题

在 HVA 到 HPA 的转换过程中，当前的实现是主动调用 get_user_pages 来分配物理页，我们又知道 VMM 运行在内核态，实际上，它是有能力直接为 GPA 分配物理内存，而不必再借助 HVA 的，那为什么 KVM 要选择保留 HVA 呢？欢迎你在留言区分享你的想法和收获，我在留言区等你。

吊打面试官

- 谈谈你对虚拟机的理解。

在计算机领域，虚拟机是一个多义词。一定要先和面试官确认，他说的虚拟机是指的KVM这种整机虚拟机，还是JVM这种语言虚拟机。

你可以从这两个角度进行回答，第一是虚拟机产生的动机，是为了更充分地利用服务器资源，虚拟机可以让运行在宿主上的各个客户端相互隔离，这样就提供了更好的安全性。第二是虚拟机的实现包含了CPU虚拟化，内存虚拟化，IO设备虚拟化，中断虚拟化等多个方面。一般情况下，面试官会从上述四个方面再挑一个进行追问。如果是内存虚拟化，那么就是这节课的内容。如果是其他方面，可能就要你自己再去深入学习，加以准备了。

高频面试真题

极客时间

好啦，这节课到这就结束啦。欢迎你把这节课分享给更多对计算机内存感兴趣的朋友。我是海纳，我们下节课再见！

上一篇 11 | 即时编译：高性能JVM的核心秘密

精选留言 (2)

写留言

梅侯
2021-11-19

Guest 可以在 root 模式下正常执行指令，就如同在执行物理机的指令一样。

这里写错了？应该是non-root模式？

大鑫仔Yeah
2021-11-19

沙发

展开