第 3 章

内存虚拟化

随着大数据时代的到来，数据处理成为云计算的核心能力。例如在大数据处理系统中，应用程序需要访问大量内存中的数据，于是低延迟内存访问在云环境中至关重要，然而云计算采用虚拟化技术，却给内存访问带来了更大开销。因此要解决云环境下的内存管理问题，就要对内存虚拟化技术有深入了解。

本章 3.1 节介绍内存虚拟化中的基本概念，使读者了解内存虚拟要解决的基本问题，以及内存虚拟化的基本原理。3.2 节介绍内存虚拟化的三种实现方式，包括软件方式、硬件辅助方式以及半虚拟化的方式，并分析每种方式的优缺点，包括可能的优化以及最新进展。3.3 节介绍 x86 架构下的内存虚拟化在 QEMU/KVM 中的实现，通过描述其中主要数据结构以及主要流程，使读者了解内存虚拟化的实现方式。3.4 节作为内存虚拟化的拓展，介绍了“多虚一”环境下 QEMU/KVM 中的实现。除了介绍基本原理和实现原理，本章还增加了相关实验和相关研究的最新进展，让读者更加深入地理解内存虚拟化知识。

3.1 内存虚拟化概述

内存是计算机系统中的重要部件。在任何广泛使用的计算机体系结构中，CPU 若没有内存提供指令、保存执行结果，则无法运行，CPU 从设备读取的数据也将无法保存。由 DRAM(Dynamic Random Access Memory，动态随机访问存储器)芯片组成的内存也是存储器层次结构中重要的一环，其速度虽然慢于 SRAM(Static Random Access Memory，静态随机访问存储器)，但因为价格更低廉，可以获得更大的容量。

在当前的生产环境中，内存容量已经达到了 TB 级别。随着大数据时代的到来，大量数据需要保存在内存中进行处理，才能够获得更低的延迟，从而缩短用户的等待时间，提升用户体验。云时代的到来为计算机内存管理提出了新的挑战：需要为客户机提供从 0 开始的、连续的、相互隔离的虚拟“物理内存”，并且使内存访问延迟低，接近宿主机环境下的内存访问延迟。这就是本章主要介绍的内容，即如何高效地实现内存虚拟化。

广义的内存虚拟化不仅包括硬件层面的内存虚拟化，还包含更多意义上的内存虚拟化。内存虚拟化即给访存指令提供一个**内存空间**，或称为**地址空间**。地址空间必须是从 0 开始且连续的，可以形象地看作一个大数组，通过从 0 开始的编号访问其中的元素，每个元素储存了固定大小的数据。由低层向高层、由简单到复杂，各类地址空间可以概括如下：

(1) 单机上的物理地址空间。对于一条访存指令，若系统中没有开启分页模式，在不考虑开启分段模式的情况下，这条指令可以访问全部的物理内存。指令访问的 PA(Physical

Address，物理地址）是从 0 开始且连续的。在计算机启动之初，BIOS（Basic Input Output System，基本输入输出系统）探测到主板内存插槽上的所有内存条，并给每个内存条赋予一个物理地址范围，最后给 CPU 提供一个从 0 开始的物理地址空间。从而每个内存条都被映射到一个物理地址范围内，软件无须知道自己访问的是哪个内存条上的数据，使用物理地址即可访问内存条上存储的数据。物理地址隐藏了内存条的相关信息，给系统提供了从 0 开始且连续的物理地址空间抽象，是一种虚拟化，如图 3-1(a)所示。除了内存条被映射到物理地址空间内，也有一些外围设备（Peripheral Devices）的内存和寄存器被映射到物理地址空间内。当 CPU 发出的访存指令的物理地址落在外围设备对应的物理地址范围内，则会将数据返回给 CPU，这称作内存映射 I/O。如果只有一层物理地址空间，则系统中只能运行一个进程，无法对 CPU 分时复用。

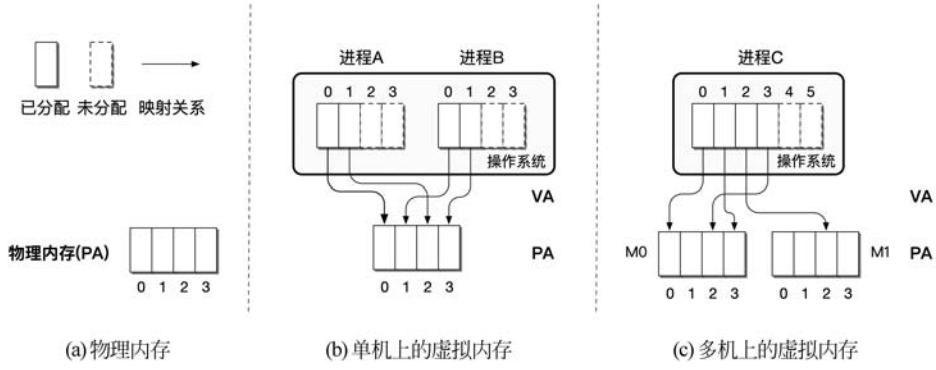


图 3-1　物理内存、虚拟内存与分布式共享内存

（2）单/多机上的虚拟地址空间。为了实现 CPU 的分时复用，操作系统提供了进程的概念。一个进程即是一串相互关联、完成同一个任务的指令序列。为了使不同进程的访存指令在访问物理内存时不会相互冲突，VA（Virtual Address，虚拟地址）的概念被引入。每个进程都有一个独立的虚拟地址空间，从 0 开始且连续，VA 到 PA 的映射由操作系统决定。这样，假设操作系统为进程 A 分配了第 0 块和第 2 块的物理内存，为进程 B 分配了第 1 块和第 3 块的物理内存，而两个进程的虚拟地址空间大小均为 4。本书使用抽象的“块”作为虚拟内存和物理内存之间映射的基本单位，即抽象层间映射的基本单位。两个进程都可以使用虚拟地址 0 访问它们拥有的第 0 块虚拟内存，而不会引起访问冲突，如图 3-1(b)所示。由于 VA 到 PA 的映射由操作系统决定，因此操作系统可以巧妙地设置该映射，使得系统中的多个进程以一种低内存占用的方式运行。假设进程 A 使用的第 2 块物理内存和进程 B 使用的第 3 块物理内存保存的数据相同，那么操作系统可以选择将进程 A、B 的第 1 块虚拟内存同时映射到系统中的第 2 块物理内存，并释放第 3 块物理内存，减少物理内存占用。减少物理内存占用的方法还有将虚拟内存块对应的物理内存换出到磁盘上，而不改变虚拟内存的抽象层，这种方式称为页换出。这就是**抽象层**提供的一个好处：给上层应用提供一个不变的“虚拟”内存，而灵活地改变其“后台”实现。虚拟内存甚至可以建立在多个物

理内存硬件上，从而实现内存资源的聚合。如图 3-1(c)所示，这种架构称为 **DSM**(Distributed Shared Memory，分布式共享内存)，它可以使单机进程无修改地运行在 M0 和 M1 上(M 代表 Machine)，3.4 节将介绍其原理，以及如何被用于实现分布式虚拟机监控器 GiantVM。除了横向扩展内存虚拟化的概念，即增加物理内存的量，还可以增加抽象层的层级数。

(3) 单机上的“虚拟”物理地址空间。如果保持物理地址空间的概念不变，继续更改物理地址空间的后台实现将会产生什么概念？一个很容易想到的想法是用虚拟内存代替物理内存条作为物理地址实现的后台。而物理内存可以提供给一整台机器使用，于是产生了内存虚拟化的概念。假设进程 A 运行在物理硬件上，它提供 4 块虚拟的物理内存给客户机 A 使用，分别对应其 0、1、2、3 块虚拟内存。客户机 A 在此“虚拟”物理内存的基础上，继续提供虚拟内存的抽象，将第 0、1 块物理内存分别给客户机中运行的进程 A1、B1 使用，分别映射到进程 A1 的第 0 块虚拟内存、进程 B1 的第 1 块虚拟内存。如图 3-2 所示，客户机进程 A 和普通进程 B 并无差别。如第 1 章所述，这里产生了 GVA、GPA、HVA 和 HPA 四层地址空间，其中 HVA 到 HPA 的映射仍然由宿主机操作系统决定，GPA 到 HVA 的映射由 Hypervisor 决定，而 GVA 到 GPA 的映射由客户机操作系统决定。这样的“虚拟”物理地址抽象有什么可利用之处呢？首先，这改变了对于访存指令的固有认知，即访存指令不一定访问真实的物理内存。只要保证虚拟硬件的抽象和原有的物理硬件相同，系统软件就可以按需灵活地更改抽象层的后台实现。如果真实的物理硬件需要更换维修，那么**虚拟机热迁移**可以将客户机迁移到另一个机器上，而不会由于更换硬件停止虚拟机的运行。这是由于“虚拟”物理内存的抽象没有改变，客户机将不会感知到它所依赖运行的硬件发生了变化。其次，根据前文对单机虚拟内存的描述，多个进程的虚拟内存总量可以超过系统上装备的

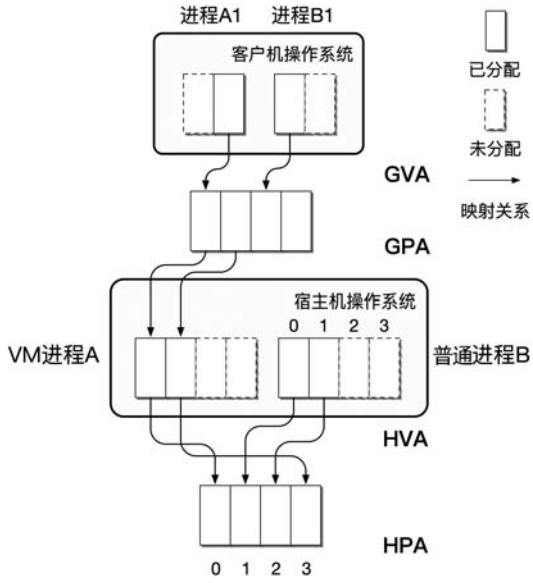


图 3-2 单机上的“虚拟”物理内存

物理内存总量。类似的，在内存虚拟化中，“虚拟”物理内存的总量可以超过真实的物理内存总量，即内存超售。

（4）多机上的“虚拟”物理地址空间。大数据环境下的应用都会占用大量的物理内存。单个机器渐渐无法满足大数据处理任务运行过程中所需要的内存空间。于是，人们将关注点从高配置的单机纵向扩展（Scale-up）转向了数量较大的单机横向扩展（Scale-out）。为了使大数据应用运行在多个机器上，而掩盖掉网络通信、容错等分布式环境下需要额外注意的复杂度，大数据框架被开发出来，如 Spark、Hadoop 等。但这些框架仍然有陡峭的学习曲线，程序员需要学习其复杂的编程模型才能在分布式框架上编写代码。如果存在一个 SSI（Single System Image，单一系统镜像），即在多个节点组成的分布式集群上给程序员提供一种单机的编程模型，则会极大地提高分布式应用的开发效率，彻底掩盖分布式系统的复杂性，无须学习分布式框架的编程模型。若把前文 DSM 的思想用于实现“虚拟”物理内存，将获得一个容量巨大的“虚拟”物理内存。DSM 在多个物理节点之上建立了一个“虚拟”物理内存的抽象层，如图 3-3 所示，M0、M1 分别配备 4 块物理内存。仿照单机上的“虚拟”物理内存，此处仍由宿主机的 VM 进程 A、B 为跨界点客户机 VM 提供“虚拟”物理内存。于是，VM 拥有了 8 块物理内存，其后台实现是两个物理机的虚拟内存。这被称为“多虚一”，即多个节点共同虚拟化出一个虚拟机。DSM 保持了“虚拟”物理内存的抽象层不变，任何一个操作系统均可运行在这样的抽象之上，和运行在真实物理硬件上没有差别。DSM 抽象层的后台实现将在 3.4 节介绍。

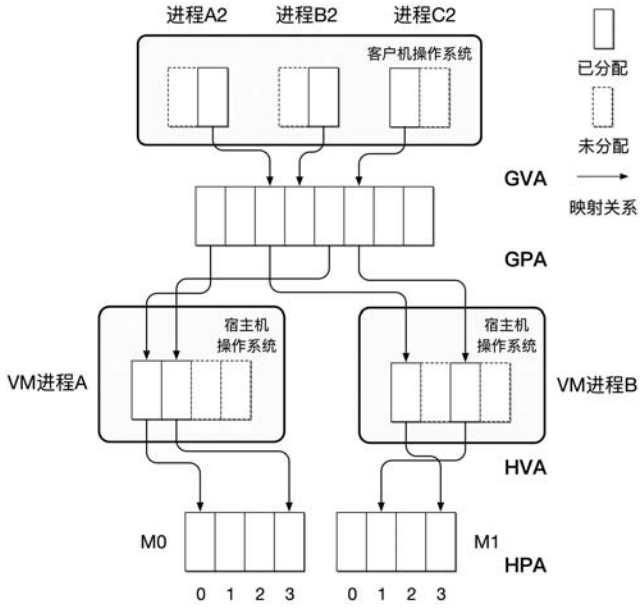


图 3-3　多机上的“虚拟”物理内存

3.2 内存虚拟化的实现

3.1 节叙述了各种各样的内存虚拟化模型，本节讨论在真实的操作系统中如何对这些内存虚拟化进行实现。将物理内存条抽象为物理地址空间由 BIOS 等实现，与本章主题不相关。下面首先介绍虚拟地址的实现，再介绍内存虚拟化的实现。作为拓展部分，建立多机上内存抽象的方法详见 3.4 节。

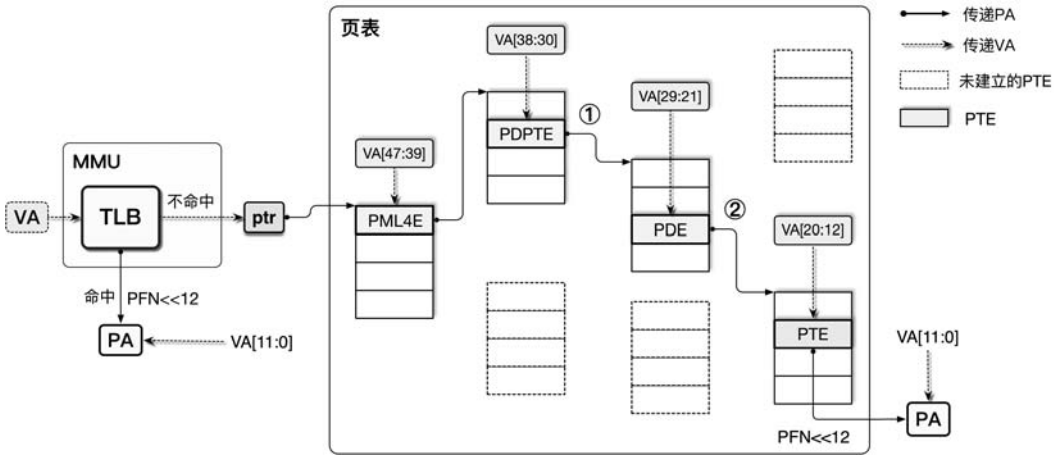
3.2.1 虚拟内存的实现：页表

如 3.1 节介绍，所有访存指令提供的地址均为 VA，还需要将 VA 转换为 PA，才能从真实的内存硬件中获取数据。如何实现 VA 到 PA 的转换？一个简单的想法是：将每个 VA 和 PA 的对应关系记录在一个表中，使用 VA 查询该表即可找到对应的 PA。这样的映射表会占用大量内存，故在现代操作系统中，虚拟内存和物理内存被分为 4KB 页，映射表中只记录 VFN(Virtual Frame Number，虚拟页号)对应的 PFN(Physical Frame Number，物理页号)，映射表表项数量减少为原来的 1/4096。映射表记录了虚拟页与物理页之间的映射，于是得名 PT(Page Table，页表)，其表项称为 PTE(Page Table Entry，页表项)。为了进一步减少 PTE 数量，也可以使用 2MB/1GB 大小的页，即大页。

所有系统软件的设计都追求时间尽可能短、空间占用尽可能少，地址翻译系统的设计也一样。事实上，虚拟地址空间十分庞大，应用程序不可能在短时间内访问大量的虚拟内存，而是多次访问某些范围内的虚拟内存，即存在**空间局部性**。基于这一观察，操作系统设计者将页表组织成基数树，或称为多级页表，可以使未使用的查询表项不出现在内存中，大大减少内存占用。在 32 位架构(如 x86)中，操作系统使用 10+10 形式的二级页表，用前 10 位索引一级页表，后 10 位索引二级页表。而在 64 位架构(如 x86-64、ARMv8-A)中，操作系统使用 9+9+9+9 形式的四级页表，如图 3-4 所示，其中页表的每一级分别用 PML4(Page Map Level 4，第 4 级页映射)、PDPT(Page Directory Pointer Table，页目录指针表)、PD(Page Directory，页目录)、PT 表示，查询一次页表需要 4 次内存访问。在 32 位系统中，当一个应用程序连续访问了连续的 4MB 虚拟内存，使用 10+10 的二级页表则仅需要 1 个一级页表页和 1 个二级页表页，页表占用内存大小为 8KB。而使用一级页表则需要 4MB 内存，仅仅是查询页表时多了一次内存访问，内存占用就减小为原来的 8KB/4MB=1/512，这是十分划算的。

查询页表不应该由应用程序负责，因为这对于应用程序完成自己的工作是无意义的，并且由应用程序负责修改页表会产生虚拟内存泄漏的问题，虚拟内存的隔离性将不复存在，造成安全问题。于是，CPU 芯片设计者引入了 MMU(Memory Management Unit，内存管理单元)负责**查询页表**。每个 CPU 核心上都配备了一个独立的 MMU，只要 CPU 将页表的基地址放入一个指定的寄存器中，MMU 中的 PTW(Page Table Walker，页表爬取器)即可查找页表将 CPU 产生的 VA 自动翻译成 PFN，左移 12 位后与 VA 的低 12 位相加即得

到 PA，不需要 CPU 执行额外的指令。这一指定的寄存器在 Intel 体系中是 CR3，在 ARM 体系中是 TTBRx(Translation Table Base Register x，翻译表基地址寄存器 x)。这些架构下的地址翻译原理大致相同，故本章统一使用 **ptr**(pointer，指针)代表页表基地址，如图 3-4 所示。同时，MMU 硬件也通过 PTE 上的标志位实现了**访存合法性的检查**，以及内存访问情况的记录。Intel 体系下的 PTE 标志位详见 Intel SDM 卷 3[①] 第 4 章，其他体系有相应的手册。本章第 3、4 节都基于 Intel x86 架构，此处列举如下 x86 架构中常用的标志位。



注：①指向 PD 页表页或 1GB 大页；②指向 PT 页表页或 2MB 大页。

图 3-4　64 位系统中虚拟地址的翻译

(1) P(Present)位。PTE[0]，置为 1 时该 PTE 有效，为 0 时该 PTE 无效。任何访问该 PTE 对应的虚拟地址的指令均引起缺页异常。当物理页未分配、页表项未建立(此时 PTE 的每一位均为 0)或物理页被换出时(此时 PTE 的每一位不全为 0)，P 位为 0，物理页不存在。

(2) R/W 位和 U/S 位。PTE[1：2]，置为 1 时表示该页是可读可写的，为 0 时表示该页只读。U/S 位置为 0 时只有管理者(supervisor，即运行在 Ring0、Ring1、Ring2 的代码)可以访问，为 1 时用户(user，即运行在 Ring3 的代码)、管理者均可访问。这两个位用于权限管理。

(3) A(Accessed)位和 D(Dirty)位。PTE[5：6]，当 CPU 写入 PTE 对应的页时 D 位被置为 1，当 CPU 读取或写入 PTE 对应的页时 A 位被置为 1，这两位均需要操作系统置 0。D 位仅存在于 PTE 中，而不存在于 PDE(Page Directory Entry，页目录项)中。D 位用于标识一个文件映射的页在内存中是否被修改，在将页换出时需要更新对应的磁盘文件。A 位用于标识内存页是否最近被访问。当操作系统将 A 位置为 0 后一段时间内，若 A 位不再变为 1，则对应的内存页不经常被访问，可以被换出到磁盘。操作系统有一套内存页换出机制，将不经常访问的内存页换出到磁盘，保证内存被充分使用。

---

① 下载网址：https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-system-programming-manual-325384.htm，请参阅最新版。

当 MMU 硬件无法完成地址翻译时，则需要操作系统软件的配合。在地址翻译的过程中，MMU 硬件仅负责页表的查询，而操作系统负责页表的维护。当 MMU 无法完成地址翻译时，就会向 CPU 发送一个信号，产生了**缺页异常**（在 x86 架构中是 14 号异常），从而调用操作系统内核的缺页异常处理函数。内核按照其需要为产生缺页异常的 VA 分配物理内存，建立页表，并重新执行引起缺页异常的访存指令；如果该访存指令访问的虚拟内存被换出到磁盘上，则需要首先分配物理内存页，再对磁盘发起 I/O 请求，将被换出的页读取到新分配的物理内存页上，最后建立对应的页表项。操作系统可以灵活地利用缺页异常，实现内存换出、COW（Copy-On-Write，写时复制）等功能。

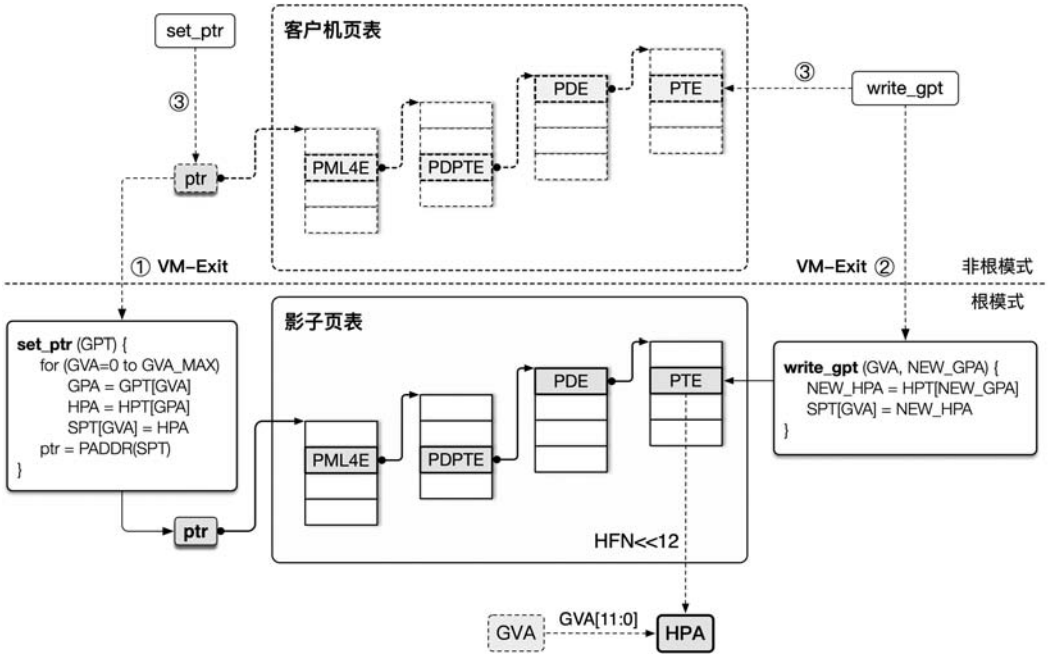
缩短时间的一种重要方式是**缓存**。在 MMU 中，**TLB**（Translation Lookaside Buffer，翻译后备缓冲器）用于缓存 VA 到 PA 的映射，避免查询页表造成的内存访问。MMU 中也有**页表缓存**，用于缓存 PTE，进一步优化 TLB 不命中时的性能，然而，内存容量随着技术的进步快速增大，TLB 容量的增速却很缓慢，这导致 TLB 能够覆盖的虚拟内存空间越来越小。研究表明，应用运行时间的 50%都耗费在查询页表上[1]。前文提到的**大页**可以在一定程度上解决这一问题，但不够灵活。Vasileios 等提出了区间式 TLB 机制[16]，每个区间 TLB 项可以将任意长度的虚拟内存区间映射到物理内存，该长度由操作系统决定，进一步减少了 PTE 的数量。这两种方式使得 TLB 能够覆盖更多的虚拟地址空间，减少了查询页表的次数。在进程切换时，由于进程运行在不同的虚拟地址空间中，需要将 TLB 中的所有项清空，否则地址翻译将出错。x86/ARM 硬件提供了 **PCID**（Process Context IDentifier，进程上下文标识符）/**ASID**（Address Space IDentifier，地址空间标识符），将 TLB 中的项标上进程 ID，于是在进程切换时无须清空 TLB。详见 Intel SDM 卷 3 第 4.10 节。

3.2.2 内存虚拟化的软件实现：影子页表

在新系统设计的过程中，复用已有设计可以加快设计的流程，降低设计的难度。是否可以重用 CPU 芯片上的 MMU 从而实现 GVA 到 HPA 的翻译呢？客户机操作系统运行在非根模式下，将客户机页表起始地址 GPA 写入 **ptr**，这样 MMU 只能完成 GVA 到 GPA 的翻译。将页表修改为 GVA 到 HPA 的映射，MMU 就能将 GVA 翻译为 HPA，并且对客户机完全透明，从而完成问题的转化。

具体操作过程是：当 vCPU 创建时，Hypervisor 为 vCPU 准备一个新的空页表。当 vCPU 将 GPT 的 GPA 写入 **ptr** 时，引起一次 VM-Exit，vCPU 线程退出到 Hypervisor 中，调用处理 **ptr** 写入的处理函数（**步骤 1**）。Hypervisor 中保存了 GPA 到 HVA 的映射，处理函数读取客户机试图写入 **ptr** 值，翻译为 HVA，即可读取 GPT 的内容。进而，Hypervisor 遍历 GPT，并将每个 GPT 表项中保存的 GPA 翻译为 HVA，再使用宿主机操作系统内核翻译为 HPA，最后将 HPA 填入新页表的 GVA 处（**步骤 2**）。完成 GPT 的遍历后，就建立了对应的新页表，Hypervisor 的处理函数最终将该新页表的基地址写入 **ptr** 中，并返回到 vCPU 重新执行引起 VM-Exit 的指令（**步骤 3**），见图 3-5 中标号①。GPT 的所有页表页被标记为只读，客户机写入 GPT 引发 VM-Exit，并调用 Hypervisor 的相关处理函数，把对

GPT 的修改翻译为对 SPT 的修改，完成新页表与 GPT 之间的同步，见图 3-5 中标号②。新页表称为**影子页表**，因为对于每个 GPT，都需要对应的 SPT 作为代替，且 SPT 与 GPT 的结构完全相同，其不同仅仅是每个表项中的 GPA 被修改为了对应的 HPA，如同影子一样。影子页表查询的结果是 HFN(Host Frame Number，宿主机页号)，左移 12 位后与 GVA 的低 12 位相加即得到对应的 HPA。



注：①写入 ptr 引起的退出；②写入 GPT 引起退出；③写入被 Hypervisor 截获。

图 3-5　使用影子页表翻译客户机虚拟地址

和非虚拟化场景下的页表相同，MMU 硬件可以自动查询 SPT，将 CPU 产生的 GVA 直接翻译为 HPA，也有 TLB 保存 GVA 到 HPA 的映射以加快地址翻译。虚拟内存的优势也可应用在 VM 进程的内存管理上，如宿主机操作系统可以将 VM 进程不常用的内存页换出到磁盘，两个客户机之间也可以通过页表的权限实现简单的隔离，还可以通过将两个内存插槽对应的 HVA 映射到相同的 HPA，应用 COW 技术实现客户机之间的内存共用，从而节约系统的物理内存，实现内存超售。

为了将 GPT 翻译为 SPT，需要利用宿主机中保存的 GPA 到 HPA 的映射。由于 HVA 到 HPA 的映射由宿主机页表保存，Hypervisor 仅需关注 GPA 到 HVA 的映射。事实上，在 Hypervisor(如 KVM)中，**内存插槽**(kvm_memory_slot)数据结构将一段 GPA 一对一映射到 HVA，用宿主机的虚拟内存作为虚拟的“内存插槽”给客户机使用。内存插槽在 HVA 中的位置可以不从 0 开始且不连续，但 Hypervisor 使用多个内存插槽可以给客户机提供一个从 0 开始且连续的物理地址空间。客户机内存插槽的设计在本章后面的小节中有详细介绍。由于 Hypervisor 需要为每个客户机中的每个进程维护一个独立的 SPT，系统中 GPT

的数目等于 SPT 的数目，这将占用大量内存。当 GPT 被修改时，由于需要保持 SPT 与 GPT 的同步，vCPU 线程需要 VM-Exit 将控制权转让给 Hypervisor，由 Hypervisor 根据 GPT 的修改来修改 SPT，同时将 TLB 中缓存的相关项清除。于是，每次客户机对 GPT 的修改都将引起巨大的性能开销，只有当客户机很少修改 GPT 时，SPT 才会表现出较好的性能。

3.2.3 内存虚拟化的硬件支持：扩展页表

仔细回顾虚拟化环境下的地址翻译，可以发现 GPA 到 HPA 的转换一步可以从影子页表中剥离出来。当客户机创建时，宿主机给客户机使用的内存已经分配完毕，在客户机的运行过程中，很少改变 GPA 到 HPA 的映射。其次，SPT 和 GPT 也包含重复的信息，即等价于包含了两次 GVA 到 GPA 的映射；两个进程的 SPT 之间也包含重复的信息，即重复多次包含了 GPA 到 HPA 的映射，故增加了页表维护的复杂度并增大了内存开销。于是，系统设计者将 GPA 到 HPA 的映射从影子页表中剥离出来，形成一个新的页表，配合 GVA 到 GPA 映射的页表(GPT)共同完成地址翻译。

虚拟环境下的地址翻译依赖于**双层页表**(Two Level Paging)。GPA 到 HPA 映射的页表在 x86 中称为 EPT(扩展页表)，在 ARM 中称为第二阶段页表(Stage-2 Page Table)，而 GPT 称为第一阶段页表(Stage-1 Page Table)，一、二阶段页表基地址分别保存在 TTBR0_EL1 和 VTTBR_EL2 中(见第 5 章)。本章使用 **gptr**(guest pointer，客户机指针)表示 GPT 基地址寄存器，HPT(Host Page Table，宿主机页表)表示被剥离出的页表；**hptr**(host pointer，宿主机指针)表示 HPT 基地址寄存器。每个客户机有一个私有的 HPT，包含与 GPT 完全不重复的信息。由于 HPT 与 GPT 之间没有依赖关系，修改 GPT 时无须修改 HPT，即无须 Hypervisor 干预，从而减少了客户机退出到 Hypervisor 的次数。

ARM 与 Intel VT 都提供了类似的双层页表支持，本章只介绍 Intel VT 中提供的支持，后文统一使用 **EPT** 表示保存了 GPA 到 HPA 映射的页表。Intel VT 提供了具有交叉查询 GPT 和 EPT 功能的扩展 MMU。当 VM-Entry 时，EPTP(EPT Pointer，扩展页表指针，即 EPT 的基地址)将由 Hypervisor 进行设置，保存在 VM-Execution 控制域中的 EPTP 字段中。需要注意的是，每个客户机中的所有 vCPU 在运行前，其对应的 VMCS 中的 EPTP 都会被写入相同的值。这是因为所有的 vCPU 应该看到相同的客户机物理地址空间，即所有 vCPU 共享 EPT。一个客户机仅需一个 EPT，故减小了内存开销。

EPT 表项的构成较为简单，其第 0、1、2 位分别表示了客户机物理页的可读、可写、可执行权限，并包含指向下一级页表页的指针(HPA)。当 EPTP(存在于 VMCS 中)的第 6 位为 1 时，会使能 EPT 的 A/D(Accessed/Dirty，访问/脏)位。EPT 中的 A/D 位和前文所述的进程页表的 A/D 位类似，D 位仅存在于第四级页表项，即 PTE 中。A/D 位由处理器**硬件**置 1，由 Hypervisor **软件**置 0。每当 EPT 被使用时，对应的 EPT 表项的 A 位被处理器置 1；当客户机物理内存被写入时，对应的 EPT 表项的 D 位被置 1。需要注意的是，**对客户机页表 GPT 的任何访问均被视为写**，GPT 的页表页对应 EPT 表项的 D 位均被处理器硬件置 1。该硬件特性将在本章多处提及，可用于实现一些软件功能，也可选择关闭该特性，例如可以

用此硬件特性实现虚拟机热迁移。详见 Intel SDM 卷 3 第 28.2.4 节。

和客户机操作系统中的 GPT 类似，EPT 也是在缺页异常中由 Hypervisor 软件建立的。当刚启动的客户机中的某进程访问了一个虚拟地址，由于此时该进程的一级页表（GPT 中的 PML4）为空，故触发客户机操作系统中的缺页异常（**步骤 1**）。客户机操作系统为了分配 GPT 对应的客户机物理页，需要查询 EPT。此时，由于 EPT 尚未建立，客户机操作系统就退出到了 Hypervisor（**步骤 2**）。当客户机操作系统访问了一个缺失的 EPT 页表项，处理器产生 EPT 违例（EPT Violation）的 VM-Exit，从而 Hypervisor 分配宿主机内存、建立 EPT 表项（步骤 3）。触发 EPT 违例的详细原因会被硬件记录在 VMCS 的 VM-Exit 条件（VM-Exit Qualification）字段，供 Hypervisor 使用。宿主机操作系统完成宿主机物理页分配，建立对应的 EPT 表项，将返回到客户机操作系统（**步骤 3**）。客户机操作系统继续访问 GPT 的下一级页表（PDPT），重复步骤 1、2，GPT 和 EPT 的建立方可完成。这里又出现了软硬件的明确分工：**软件**维护页表，**硬件**查询页表。如果访问了 EPT 中的一个配置错误，不符合 Intel 规范的表项，处理器会触发 EPT 配置错误（EPT Misconfiguration）的 VM-Exit。例如访问了一个不可读但可写的表项，此时硬件将不会记录发生 EPT 配置错误的原因，这类 VM-Exit 被 Hypervisor 用于模拟 MMIO。有关 EPT 硬件的详细内容请参阅 Intel SDM 卷 3 第 28 章，Hypervisor 软件部分的介绍见本书 3.3 节。

当处在非根模式下的 CPU 访问了一个 GVA，MMU 将首先查询 GPT。**gptr** 包含客户机页表的起始地址 GPA，这会触发 MMU 交叉地查询 EPT，将 **gptr** 中包含的 GPA 翻译成 HPA，从而获取 GPT 的第一级表项。同理，为了获取 GPT 中每个层级的页表项，MMU 都会查询 EPT。在 64 位的系统中，Hypervisor 使用 GPA 的低 48 位查询 EPT 页表，而 EPT 页表也使用了 9＋9＋9＋9 的四级页表的形式。假设 GPT 也是四级页表，那么非根模式下的 CPU 为了读取一个 GVA 处的数据，如图 3-6 所示，需要额外读取 **24 个页表项**（图中加粗

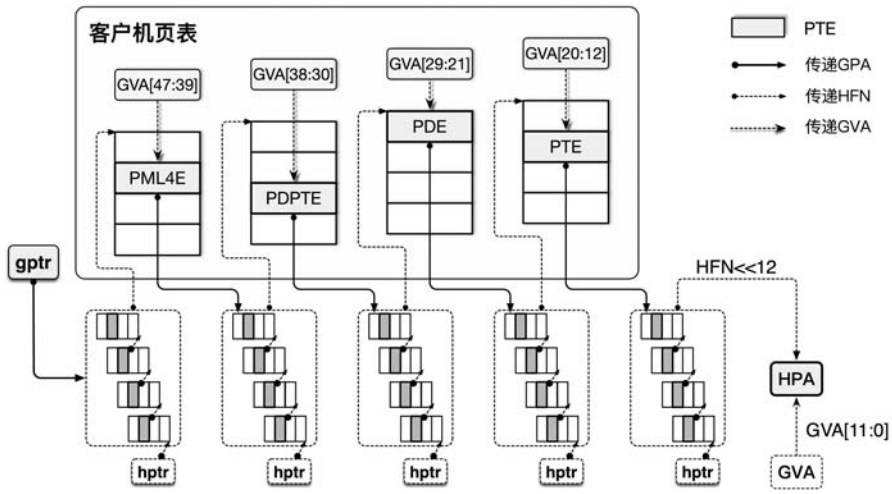


图 3-6　使用双层页表翻译客户机虚拟地址

黑框的灰色长方形），因此需要额外的 24 次内存访问。即使 MMU 中有 TLB 缓存 GVA 到 HPA 的映射，TLB 也无法覆盖越来越大的客户机虚拟地址空间，双层页表的查询将会造成巨大的开销。而在 TLB 未命中时，一个四级影子页表仅需访问 4 次内存即可得到 HPA，相比于双层页表有巨大的优势。是否可以将影子页表与扩展页表相结合呢？

3.2.4 扩展页表与影子页表的结合：敏捷页表

在系统设计中，经常存在着各种各样的折中与权衡。虽然影子页表和双层页表（即 x86 中的扩展页表）在 TLB 命中时，均可以以最快的时间获得 GVA 到 HPA 的映射，但遭遇 1 次 TLB 未命中时，查询双层页表需要访问内存的次数增长到了 24/4＝6 倍。然而影子页表会引起大量的 VM-Exit，尤其是在频繁分配、释放虚拟内存的内存密集型场景下，这使得影子页表在现在的虚拟化环境下很少使用。由于内存容量快速增大、TLB 容量增长缓慢，TLB 不命中的次数越来越多，查询双层页表造成的 6 倍内存访问也造成了不可忽视的开销，影子页表有一些优势。表 3-1 将影子页表与扩展页表进行了对比。

表 3-1 影子页表与扩展页表对比

对比项	硬件扩展	TLB 功能	查询访存次数	更改 GPT	内存占用
PT	否	VA→PA	4	无须退出	低
SPT	否	GVA→HPA	4	需要退出	高
EPT	是	GVA→HPA	24	无须退出	低

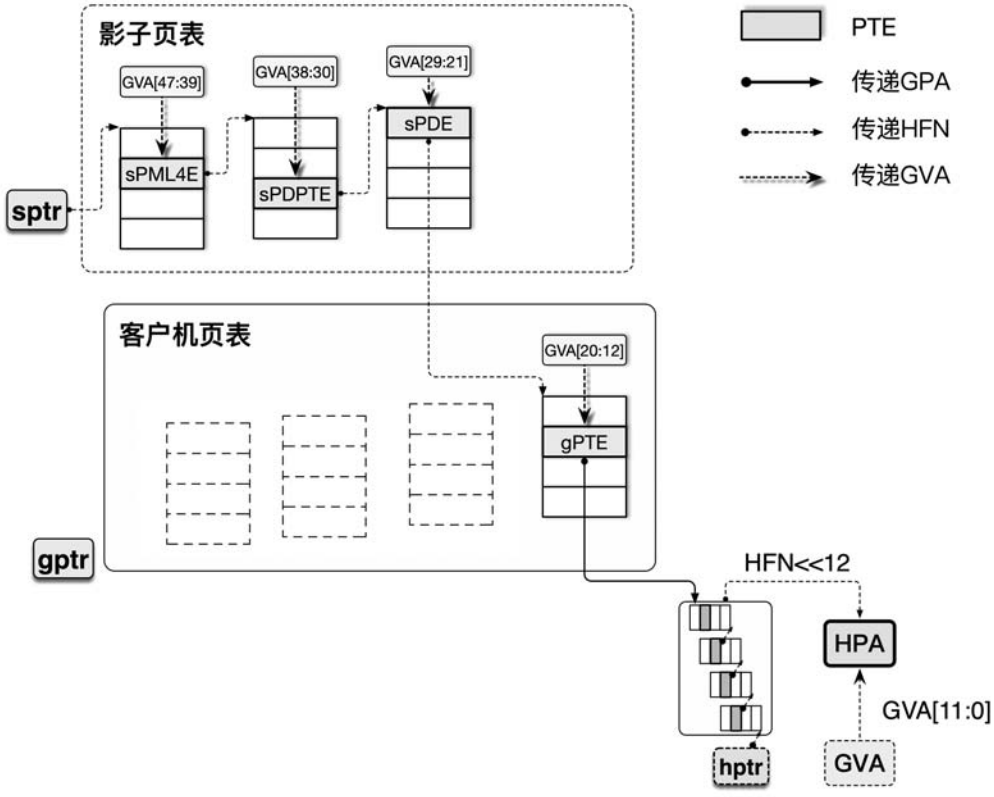
Jayneel 等人观察到[1]，在 2 秒的采样时间内，仅有 1～5％的地址空间被频繁修改，且被修改的地址空间会比未修改的地址空间修改得更加频繁。例如，保存代码的地址空间极少被修改、写入，称为**静态区**；而地址空间的堆栈以及映射文件的部分则被频繁修改，称为**动态区**。若用 SPT 完成静态区的地址翻译，则能减小 TLB 不命中时的页表查询开销；而用双层页表完成动态区的地址翻译，则能避免 GPT 频繁修改造成的 VM-Exit。于是，研究人员于 2016 年提出了敏捷页表[1]。他们设计了一种影子页表和双层页表混用的机制，还有一种策略决定何时由影子页表转换到双层页表或由双层页表转换到影子页表。这是一种机制与策略的分离：机制是固定的，而程序员可以灵活修改策略，具有更好的灵活性。图 3-7 展示了影子页表与双层页表的混用机制。为了实现敏捷页表，需要硬件支持以及软件支持，下面分开介绍。

先介绍影子页表与双层页表的混用机制，这部分功能主要使用硬件实现。首先，影子页表需要提供转换位（Switching Bit）的支持。在影子页表的页表项中，仍有一些标志位被硬件忽略，可以放置转换位。硬件增加了 **sptr**（Shadow Pointer，影子指针）寄存器，用于放置影子页表基地址；同时保留原有的 **ptr** 寄存器（更名为 **gptr** 寄存器），用于放置客户机页表基地址；**hptr** 放置 EPT 的基地址，用于查询 EPT。当查询 SPT 时读到一个转换位被置为 1 的影子页表项，则 MMU 切换到双层页表的地址翻译模式继续交叉地查询 GPT 和 EPT。在切换前后，MMU 中的 TLB 仍然保存了 GVA 到 HPA 的映射，无须刷新 TLB。在

转换位被置为 1 的影子页表项中，记录了下一级 GPT 页表页的起始地址。该地址为 HPA，转换位被置为 1 时由硬件写入。通过该 HPA 即可查询到下一级 GPT 的页表项，使得页表查询过程继续进行下去。

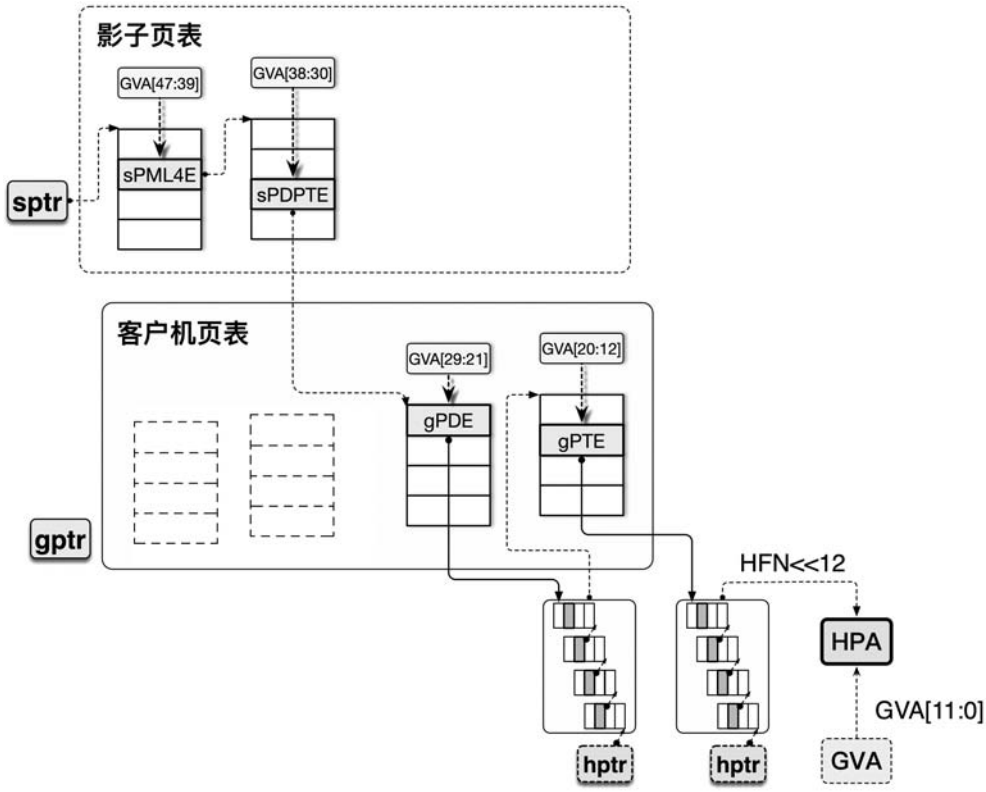
在影子页表中，可以将任何一级页表项的转换位置为 1，如图 3-7(a)中 SPT 的第三级页表项的转换位置为 1，图 3-7(b)中 SPT 的第三级页表项的转换位置为 1。图 3-7(a)中查询 3 级影子页表，仅额外读取 3＋1＋4＝**8 次**页表项(图中的加粗黑框灰色长方形)；图 3-7(b)中仅遍历 2 级影子页表，但要额外读取 2＋2＋8＝**12 次**页表项。

转换位是否置 1 由 Hypervisor 决定，这属于策略设计需要关注的。当 MMU 进入双层页表查询模式后，其读取的 GPT 表项不再被标记为只读，可以被客户机操作系统写入而不引起 VM-Exit。为了利用这一优势，Hypervisor 实现了一套策略维护转换位。当客户机进程被创建且调度运行时，MMU 处在影子页表地址翻译的状态，GPT 被标记为只读。若一个 SPT 页表项被修改，将发生 VM-Exit，Hypervisor 记录 1 次对该 SPT 页表项的修改并更新 SPT，返回客户机。当 Hypervisor 记录到 2 次对该 SPT 页表项的修改时，则将该 SPT 页表项的转换位置为 1，表明该 SPT 页表项经常被修改，需要切换到双层页表模式。如



(a) 需要8次内存读取的敏捷页表

图 3-7　使用敏捷页表翻译客户机虚拟地址



(b) 需要12次内存读取的敏捷页表

图 3-7 （续）

图 3-7(a)所示，当客户机修改了两次 sPDE(shadow Page Directory Entry，影子页目录项)时，将 sPDE 的转换位置为 1，那么对于 gPTE(guest Page Table Entry，客户机页表项)的修改将不会引起 VM-Exit，代价只是由读取 4 次内存增加到读取 8 次。

还需要一套策略决定转换位置 0。当客户机页表很少被修改时，如果将 MMU 地址翻译模式部分切换回 SPT 的翻译模式，在 TLB 不命中时可以减少读取页表项的次数。然而，假设此时有对 GPT 和 EPT 频繁的只读访问，客户机将不退出 Hypervisor，Hypervisor 也无法得知何时应该切换回影子页表模式。为此，前文所述的 EPT A/D 位特性(见 Intel SDM 卷 3 第 28.2.4 节)应该被关闭，因为当 EPT A/D 位使能时，所有对 GPT 的访问无论读/写均视为写(注意，除 GPT 页表页的只读访问并不被视为写)，于是 GPT 所在的客户机物理页对应的 EPT 表项中脏位(即 D 位)将被置 1。此时 Hypervisor 可以通过 EPT 中的脏位观察是否应该回到 SPT 模式。在一个检查周期开始时，将所有 GPT 对应的 EPT 表项脏位置 0；在周期结束时，若脏位仍然没有置 1，则视为 GPT 修改不频繁，对应的转换位置为 0，切换回双层页表的翻译模式。

敏捷页表仅是一个设想，所需的硬件支持尚未实现。研究者用仿真工具模拟了敏捷页

表的运行，并测得相比于双层页表更低的 TLB 不命中开销，以及相比于影子页表更少的 VM-Exit 次数。然而，敏捷页表对进程切换不友好，由于敏捷页表的一级页表使用了影子页表的页表页，在切换 **sptr** 时会引起 VM-Exit，但有相应的硬件优化，详见参考文献[15]。

3.2.5　内存的半虚拟化：直接页表映射与内存气球

上文所述的内存虚拟化实现均基于一个前提：客户机**无从得知**自己使用的是“虚拟”的物理内存。如果客户机知道自己运行在“虚拟”的内存硬件上，内存虚拟化的实现是否会更简单？内存虚拟化实现的性能是否更高？本节介绍两个与内存半虚拟化的相关技术。

（1）**直接页表映射**。半虚拟化可以通过告知客户机运行在虚拟环境下，让客户机**协同** Hypervisor 完成虚拟化任务，从而可以使 Hypervisor 需要完成的工作更少，Hypervisor 的实现更为简单。将半虚拟化的思想应用在内存虚拟化上，则 Hypervisor 有能力告知客户机操作系统：将页表维护成能够直接安装到真实硬件 MMU 的版本，Hypervisor 将不对客户机页表进行任何修改。GPT 中将保存 GVA 到 HPA 的映射，而 Hypervisor 需要做的仅仅是告知客户机操作系统可以使用的真实物理内存范围。这样，只需要增加客户机操作系统的一些复杂性，就不需要降低客户机运行性能的影子页表，也不需要复杂的 EPT 硬件扩展，内存虚拟化的困难减小，性能也得到提高。这种内存虚拟化的实现方式称作**直接页表映射**。

然而，由于页表中的映射对于客户机之间的隔离性、系统安全等至关重要，客户机对页表不可随便更改。在客户机更改页表时，只能调用 Hypervisor 提供的超级调用，由 Hypervisor 检查客户机映射的内存范围是否合法，才能返回客户机继续执行。相比于影子页表，Hypervisor 需要完成复杂的 GPT 到 SPT 的翻译，直接映射大大减轻了 Hypervisor 的负担。为了减小多次非根模式与根模式切换带来的开销，客户机可以选择将多次对 GPT 的更改组合起来，合并成一次超级调用进入 Hypervisor，从而将多次 CPU 模式切换替换为 1 次模式切换。虽然 Hypervisor 进行了 GPT 修改的合法性检查，但由于客户机明确地知道真实物理硬件的物理地址（HPA），仍然可以利用 HPA 发起行锤（Rowhammer）攻击。该攻击具体原理如下：由于 DRAM 不断发展，厂商将 DRAM 的单元做得越来越近，而相邻单元的相互影响也越来越大，不断访问某个地址的物理内存，即可造成相邻位置内存位的翻转。若客户机得知了真实的物理内存地址，则可以对不属于自己的相邻物理内存发起行锤攻击。

（2）**内存气球**。根据对 SPT 原理的介绍，客户机的“虚拟”物理内存的后台实现其实是宿主机进程的虚拟内存，可以使用宿主机虚拟内存的功能管理客户机物理内存。例如，为了实现内存超售，给所有客户机分配的物理内存总量可以大于物理硬件的内存容量。由于抽象层这一概念的存在，系统软件的设计者可以灵活更改“虚拟”物理内存的后台实现，将“虚拟”物理内存对应的宿主机虚拟内存**换出**到磁盘，或映射到同一块宿主机物理内存，从而减小宿主机物理内存的压力。然而，Hypervisor 在决定换出哪块“虚拟”物理内存时，无法精确地得知哪些部分在未来一段时间内不会被客户机使用。即使开启了 EPT 的 A/D

位，Hypervisor 也仅仅能够得知在过去一段时间内，客户机访问了哪些页、长时间未访问哪些页，而无法得知这些页之间的关联与意义，即所谓的**语义鸿沟**。这会导致“虚拟”物理内存换出的太多或太少：“太多”会使客户机不断等待“虚拟”物理内存从磁盘换入内存，降低客户机性能；而“太少”则使 Hypervisor 没有释放那些完全可以被立即释放的“虚拟”物理内存，造成系统内存资源的浪费。

Hypervisor 无法实现高效的客户机内存换出策略的原因是：Hypervisor 无法得知客户机内部发生了什么。而半虚拟化可以很好地解决该问题，可以使客户机和 Hypervisor 更好地沟通。**内存气球**利用了客户机内核提供的物理内存分配函数，来实现客户机内存的高效释放。其主要工作流程是，Hypervisor 调用客户机提供的内存释放接口，请求客户机释放其占用的“虚拟”物理内存。客户机收到该请求后，调用其内核提供的物理内存分配函数（如 Linux 内核中的 alloc_pages 函数），并把分配好的“虚拟”物理内存范围返回给 Hypervisor。Hypervisor 可以将该“虚拟”物理内存对应的虚拟内存释放，减轻系统内存压力。由于客户机内核的物理内存分配函数会“自动”找出未被使用的物理内存，因此这种方式很简易地找出了客户机中不被使用的物理内存，大大简化了内存气球的实现。

3.3 QEMU/KVM 内存虚拟化源码

本节将深入分析 QEMU/KVM 内存虚拟化相关代码，其中 KVM 代码来自 Linux 内核 v4.19，QEMU 代码版本为 4.1.1。下文将围绕实现所需的数据结构以及相关函数进行介绍，忽略错误处理等代码，给出充足的注释，使读者易于理解。

如 3.1 节中单机上的“虚拟”物理内存所述，内存虚拟化的核心是使用虚拟内存代替物理内存条，作为“虚拟”物理内存的实现“后台”，从而给客户机提供**从 0 开始且连续**的“虚拟”物理内存。客户机访存指令提供的地址是 GVA，被宿主机 MMU 翻译成 HPA，再发送到物理内存上读取/写入数据。Hypervisor 和操作系统维护页表，将页表装载到 MMU 中，与 MMU 硬件协同完成内存虚拟化。

对应到广泛使用的 Type II Hypervisor QEMU/KVM 中，QEMU 负责在宿主机用户态分配虚拟内存，作为客户机“虚拟”物理内存的后台实现，即完成所有物理内存硬件的功能；而 KVM 负责在内核态维护 GVA 到 HPA 的映射，即维护页表，并将页表装载到 MMU 中完成软硬件的配合。这属于一种**策略**和**机制**的分离，其中 KVM 提供了地址翻译机制，而 QEMU 决定如何利用 KVM 的地址翻译机制完成内存虚拟化，实现一套功能完整的内存虚拟化策略。这种分离的好处在于，Hypervisor 的编写者可以灵活地变更策略的实现，而机制无须修改。下面分别对 QEMU 的物理内存模拟和 KVM 的页表维护进行分析。

3.3.1 QEMU 内存数据结构

为了正确地运行客户机，QEMU 需要模拟 3.1 节所述物理地址空间的所有功能。①QEMU 作为宿主机上的用户态进程，在宿主机上分配一段虚拟地址提供给客户机作为客

户机物理内存使用。②QEMU 需要模拟物理地址空间中外围设备对应的 MMIO 部分，通过截获对该内存区域的访问，完成对设备功能的模拟，使得客户机像在真实环境中一样完成 MMIO。

1. “虚拟”物理内存的分配

本节从解决第一个问题开始，即：QEMU 进程如何分配虚拟内存作为客户机的物理内存。熟悉 C 语言标准库的读者知道，要分配一段大小不固定的虚拟内存，应该调用 malloc 函数。系统首先分配足够的堆内存给 malloc 函数使用，当分配的堆内存用完时，malloc 函数调用 brk 函数修改内核中的 brk 指针，增大分配的堆内存。如果 malloc 函数请求的内存大小超过 128KB，则会调用 mmap 系统调用在虚拟内存的内存映射区而非堆上分配内存。由于 QEMU 需要给客户机分配较大块的虚拟内存作为“虚拟”物理内存，故 QEMU 选择使用 mmap 函数。mmap 函数建立的虚拟内存映射根据分配的虚拟内存是否关联到磁盘文件分为文件映射和匿名映射，此处只关注匿名映射。RAMBlock 方便了宿主机虚拟内存的管理，简称 **RB**，其定义如下。

qemu-4.1.1/include/exec/ram_addr.h

```
struct RAMBlock {
    struct MemoryRegion * mr;        // 对应的 MemoryRegion
    uint8_t * host;                  // 宿主机虚拟地址 HVA
    // 客户机物理地址相关数据 GPA
    ram_addr_t offset;
    ram_addr_t used_length;
    ram_addr_t max_length;
    char idstr[256];                 // RAMBlock 名称，在 vmstate_register_ram 函数中填充
    QLIST_ENTRY(RAMBlock) next;      // 指向 ram_list.blocks 中的下一个元素
    int fd;                 // 对应的文件描述符，当使用磁盘文件映射时使用，最终传入 mmap 函数

    unsigned long * bmap;            // 脏页位图
    uint32_t flags;                  // 标志位，如 RAM_MIGRATABLE 标记该 RAMBlock 可以被迁移
};
```

其中 host 指针保存了 mmap 函数返回的宿主机虚拟地址，max_length 保存了 mmap 函数申请的虚拟内存大小，idstr 保存了该 RB 的名称，mr 保存了其所属的 MemoryRegion。next 指向该 RB 在全局变量 ram_list 中的下一个 RB。ram_addr_t 类型代表了所有内存条组成的 GPA 空间，ram_list. blocks 将所有“虚拟”物理内存块 RB 组织在一起，根据 max_length 从大到小排列，如图 3-8 所示。

其中，offset 是在 ram_addr_t 地址空间中的偏移，used_length 是当前使用的长度，即包含有效数据的长度，max_length 是 mmap 函数分配的长度，即最大可以使用的长度。和 mmap 函数的映射类型相同，RB 也分为匿名文件对应的类型（其 fd 为－1）以及磁盘文件对应的类型（如果使用 QEMU 的-mem-path 选项）。qemu_ram_alloc_ * 函数族负责分配新的 RB，它们最终都调用 ram_block_add 函数填充 RB 数据结构，代码如下。

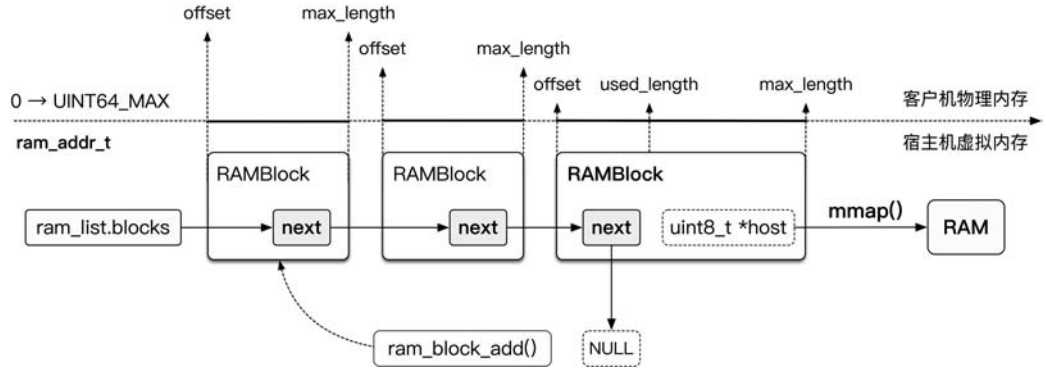


图 3-8 RAMBlock 组织结构

qemu-4.1.1/exec.c

```
static void ram_block_add(RAMBlock * new_block, Error ** errp, bool shared)
{
    new_block->offset = find_ram_offset(new_block->max_length);  //查找空位
    if (!new_block->host)
        new_block->host = phys_mem_alloc(new_block->max_length,  //调用 mmap 函数
                                         &new_block->mr->align, shared);
    .. QLIST_INSERT_BEFORE_RCU(...);                              //加入 ram_list.blocks
    smp_wmb();
    ram_list.version++;

    if (new_block->host)
        qemu_madvise(new_block->host,
        new_block->max_length, QEMU_MADV_HUGEPAGE);
}
```

参数 new_block 表示待填充的 RB。首先调用 find_ram_offset 在全局 ram_list.blocks 中查找能够容纳下 max_size 大小 RB 的位置，并将该位置填入 RB 的 offset 中。然后调用 phys_mem_alloc 函数，它最终调用 mmap 函数从而系统调用完成虚拟内存的分配，并将分配的虚拟内存起始地址填入 host 成员中。

当 new_block 完成分配后，还需要将 new_block 加入 ram_list.blocks 中，通过 QLIST_INSERT_* 函数完成。ram_list.blocks 将整个虚拟机的所有“内存条”RB 管理起来，形成了 ram_addr_t 类型的地址空间，表示所有“虚拟”物理内存条在客户机物理地址空间中所占的空间。管理 RB 的接口一般命名为 qemu_ram_*，见 exec.c 文件。最终，QEMU 调用 qemu_madvise 函数建议对该 RB 对应的虚拟内存使用大页，根据前文分析，使用大页有助于提高 TLB 命中率。ramlist 的类型 struct RAMList 如下。

qemu-4.1.1/include/exec/ramlist.h

```
typedef struct RAMList {
    RAMBlock * mru_block;            // 最近使用的 RAMBlock
    QLIST_HEAD(, RAMBlock) blocks;   // ram_addr_t 空间的链表
```

```
    // 脏页位图,用于实现 VGA、TCG、热迁移,管理粒度为一个页,即 4KB
    DirtyMemoryBlocks * dirty_memory[DIRTY_MEMORY_NUM];
    uint32_t version;                // 在 RAMList 被修改并调用 smp_wmb()后加 1
} RAMList;
```

ram_list 将 ram_list.blocks 封装成 struct RAMList 数据结构从而方便管理,是 exec.c 文件中的全局变量,保存客户机所有物理内存条的信息。其成员如下：mru_block 保存了最近使用的 RB,作为查找 ram_list.blocks 的缓存,无须遍历链表。dirty_memory 是整个 ram_list.blocks 中所有 RB 的脏页位图,每一位代表了一个脏的物理内存页,而 RB 的 bmap 位图是其一部分。为了模拟 VGA(Video Graphics Array,显示绘图阵列,可看作一种设备),QEMU 需要重绘脏页对应的界面；为了模拟 TCG(Tiny Code Generator,微码生成器,支持 QEMU 的二进制代码翻译,一种基于纯软件的虚拟化方法),QEMU 需要重新编译自调整的代码；对于热迁移,QEMU 需要重传脏页。QEMU 调用 ioctl(KVM_GET_DIRTY_LOG) 函数从 KVM 中读取脏页位图。

2. 支持“虚拟”物理内存访问回调函数

物理地址空间不仅被内存条所占据,也被外围设备的 MMIO 区域所占据,QEMU 需要对客户机访问 MMIO 进行模拟。CPU 使用 PIO 访问端口地址空间,QEMU 也需要对这类访问进行模拟。对于这些地址空间段,QEMU 无须为其分配宿主机虚拟内存,只需设置对应的回调函数。为此,QEMU 在 RAMBlock 的基础上加了一层包装,形成了 MemoryRegion,简称 **MR**,包含 MR 和回调函数。MR 代表客户机的一块具有特定功能的物理内存区域,定义如下。

qemu-4.1.1/include/exec/memory.h

```
struct MemoryRegion {
    Object parent_obj;                  // 父对象

    bool ram;                           // 是否是 RAM
    bool read_only;                     // 是否只读
    bool rom_device, ram_device;        // ROM 和 RAM 设备
    bool terminates;                    // 是否为叶子节点 MR
    bool enabled;                       // 是否使能,被注册到 KVM 中,若不使能,则在处理中忽略
    bool nonvolatile;                   // 是否是非易失性内存(non-volatile memory)

    RAMBlock * ram_block;               // 对应的 RB
    const MemoryRegionOps * ops;        // 回调函数
    MemoryRegion * container, * alias;  // 指向容器 MR、别名 MR
    Int128 size;                        // MR 大小
    hwaddr addr;                        // MR 起始地址
    hwaddr alias_offset;                // 在别名 MR 中的偏移
    int32_t priority;                   // 优先级

    QTAILQ_HEAD(, MemoryRegion) subregions;          // 容器 MR 的子 MR 链表头
```

```
    QTAILQ_ENTRY(MemoryRegion) subregions_link;     // 此 MR 在 MR 链表中对应的节点
    const char *name;                               // MR 名称,方便调试
};

struct MemoryRegionOps {
    uint64_t ( *read)(void * opaque, hwaddr addr, unsigned size);
    uint64_t ( *write)(void * opaque, hwaddr addr, uint64_t data, unsigned size);
};
```

根据本书第 4 章，QEMU 实现了 MMIO 的模拟。为此，将一个 RB 和包含了 MMIO 模拟函数的 MemoryRegionOps 绑定起来，就形成了 MR 这种表示多个种类物理内存块的数据结构。当 KVM 中表示内存条的 MR 其 ops 域为 NULL 时，ram_block 不为 NULL；而对于表示 MMIO 内存区的 MR，其 ops 注册为一组 MMIO 模拟函数时，ram_block 为 NULL。当客户机访问了一个 MMIO 对应的区域，KVM 将退出到 QEMU，调用 ops 对应的函数。ops 中包含了 read、write 等函数，其参数包括相对于 MR 的 hwaddr 地址 addr、写入的数据以及数据的大小，模拟硬件 MMIO 读写（例如 PCI Device ID（Peripheral Component Interconnect Device IDentifier，外设部件互联设备标识符））的 read 函数应该返回设备 ID。至此，QEMU 将整个物理地址空间用 MR 占满，这种既包含内存条区域又包含 MMIO 区域的物理地址空间的类型是 hwaddr。MR 的 addr 域即为 hwaddr 类型，表示该 MR 的 GPA。

QEMU 对象模型为 MR 提供了构造函数和析构函数，分别在一个 MR 实例创建和销毁时调用。在 MR 的 parent_object 销毁时，就会调用 MR 的析构函数。MR 创建时调用 memory_region_initfn 函数初始化 MR，包括将 enable 置为 true 和初始化 ops、subregions 链表等。调用 memory_region_ref 函数使 MR 的 parent_obj 的引用数加 1，memory_region_unref 函数则使 MR 的 parent_obj 的引用数减 1，若引用计数为 0，则会调用 memory_region_finalize 函数完成 MR 的析构，如果是 RAM 类型的 MR，还会释放对应的 RB。

QEMU 给所有种类的 MR 都提供了进一步封装的构造函数，根据 MR 的类型填充数据结构。这些函数是 memory_region_init_ * ，* 代表类型，下面举例介绍。

（1）RAM 类型 MR 需要调用前文的 qemu_ram_alloc 函数分配一个 RB 填入 ram_block 域，ram 域为 true；ROM 类型 MR 则需要额外将 read_only 域置为 true，表示只读的内存区。

（2）MMIO 类型 MR 负责实现 MMIO 模拟，需要传入 ops 进行初始化，其中 ops 是一组回调函数，当 QEMU 需要模拟 MMIO 时，会调用 ops 中的函数进行 MMIO 模拟。

（3）对于 ROM Device（Read Only Memory Device，只读内存设备）类型 MR，对它进行读取则等同于 RAM 类型的 MR，而写入则等同于 MMIO 类型的 MR，调用回调函数 ops。

（4）IOMMU（Input/Output Memory Management Unit，输入输出内存管理单元）类型 MR 将对该 MR 的读写转发到另一 MR 上模拟 IOMMU。所有的 MR 构造函数 memory_region_init_ * 都要调用 memory_region_init 函数填充 size、name 等成员。这些 MR 均称

为实体 MR，terminates 为 true。前三类实体 MR 的创建过程如图 3-9 所示。

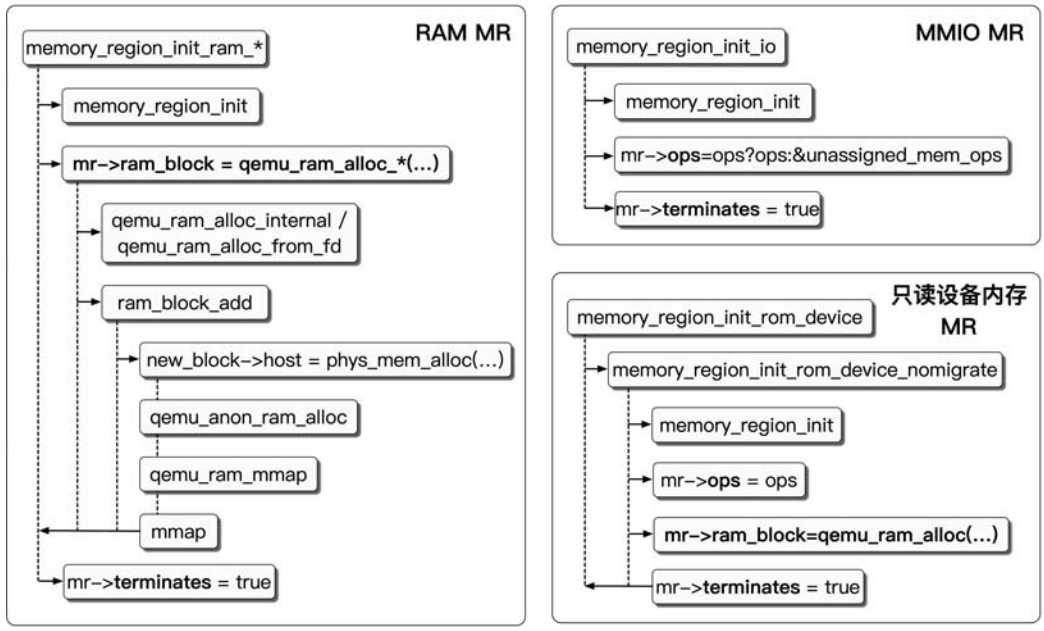


图 3-9　实体 MR 的构造函数及其调用链

所有的 MR 组成了一棵树，其叶子节点是 RAM 和 MMIO，而中间节点代表总线（容器 MR）、内存控制器（别名 MR）或被重定向到其他内存区域的内存区，树根是一个容器 MR，如图 3-10 所示。这棵树表示一个地址空间，被 AddressSpace 数据结构指向。下面介绍容器 MR 和别名 MR。

（1）容器（Container）MR 将其他的 MR 用 subregions 链表管理起来，用 memory_region_init 函数初始化。例如 PCI BAR（PCI Base Address Register，PCI 基地址寄存器）的模拟由 RAM 和 MMIO 部分组成，需要容器类型的 MR 表示 PCI BAR，通过 memory_region_add_subregion 函数加入容器 MR，子 MR 的 container 成员指向其容器 MR。通常情况下，一个容器 MR 的子 MR 不会重叠，当在该容器 MR 内解析一个 hwaddr 时，只会落入一个子 MR。但一些情况下子 MR 会重合，这时使用 memory_region_add_subregion_overlap 函数将子 MR 加入容器 MR，而解析地址时由优先级决定哪个子 MR 可见。没有自己的 ops/ram_block 的容器 MR 称为纯容器 MR，容器 MR 也可以拥有自己的 MemoryRegionOps 以及 RAMBlock。在一个容器 MR 中，subregions 链表上的所有子 MR 按照各自的优先级从小到大排列。memory_region_add_subregion_overlap 函数可以指定子 MR 的优先级，如果优先级为负，则隐藏在默认子 MR 之下，反之在上。

（2）别名（Alias）MR 指向另一个非别名类型的 MR，QEMU 通过将多个别名 MR 指向同一个 MR 的不同偏移处，从而将此 MR 分为几部分。alias 域指向原 MR（在代码中多用 orig 表示），alias_offset 是别名 MR 在实际 MR 中的位置。别名 MR 用 memory_region_

init_alias 函数初始化。别名 MR 没有自己的内存，没有子 MR。别名 MR 本质上是一个实体 MR(或另一个别名 MR)的一部分，而容器 MR 的子 MR 并非容器 MR 的一部分，仅被容器 MR 管理。别名 MR 并非容器 MR 的子 MR。3.3.2 节的实验将介绍别名 MR 以及容器 MR 的实例，介绍 QEMU 如何使用它们。

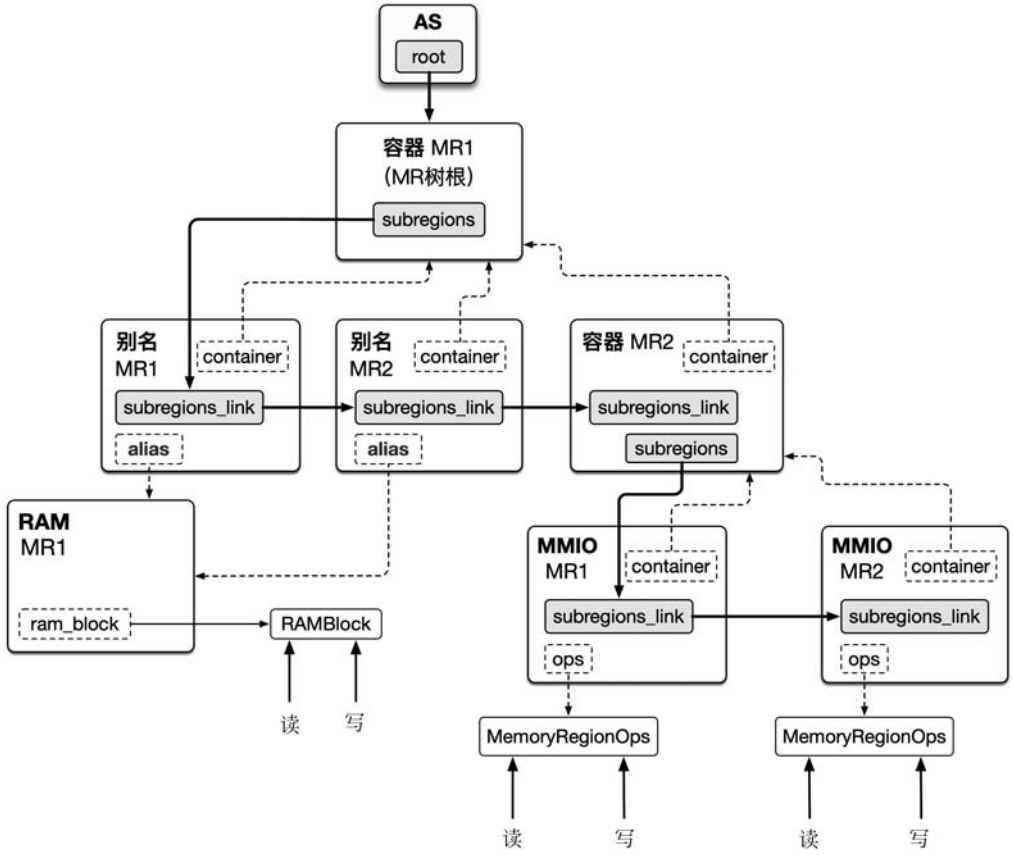


图 3-10 MemoryRegion 树

在 QEMU 中，全局变量 system_memory 是整个 MR 树的根。给定 MR 树的根 MR 以及要查询的客户机物理地址，QEMU 就可以查找该地址落在哪个实体 MR 中。首先判断该地址是否在根管理的范围内，如果不在则返回，否则进行如下步骤的搜索：①按照优先级从高到低的顺序遍历该容器 MR 的 subregions 链表；②如果子 MR 是实体 MR，且该地址落在子 MR 的[mr-> addr，mr-> addr＋mr-> size)中，则结束查找，返回该实体 MR；③如果子 MR 是容器 MR，则递归调用步骤①；④如果子 MR 是别名 MR，则继续查找对应的实际 MR；⑤如果在所有的子 MR 中都没有查找到，则查询该地址是否在容器 MR 自己的内存范围内。

想了解更完整的说明，可以查看 QEMU 源码树中的 docs/devel/memory.rst 文档，或

查阅注释[①]，其中包含 MemoryRegion 的所有概念，以及 MR 相关接口的详细解释。

3. 顶层数据结构 AddressSpace

在前几节中，QEMU 使用 mmap 函数分配了宿主机虚拟内存，作为客户机“虚拟”物理内存的实现后台，又构建了 MemoryRegion 树，对一个物理地址空间内各个不同区域的功能进行了模拟，包括内存条 RAM 以及 MMIO，使用容器 MR 对子 MR 进行管理，完成了 QEMU 对各段内存功能的模拟。前文由底层向上层，介绍了 QEMU 中抽象程度较高的数据结构 MR。进一步地，除了对客户机物理地址空间的模拟，MR 树可以用在对任何地址空间的模拟上。计算机硬件中还存在以下地址空间，与内存地址空间类似。

(1) CPU 地址总线传来的地址可以用于访问 RAM 或 MMIO，即形成了物理地址空间，3.2.5 节的 system_memory 即表示此空间。

(2) 除了 MMIO，CPU 与外围设备打交道的另一种方式是 PIO，CPU 使用 in * /out * 指令访问设备端口，端口号组成了另一种地址空间，在 x86 上有 65536 个端口，端口地址空间范围是[0, 0xffff)。

(3) 除了 CPU 可以访问内存，外围设备也可以自发地访问内存，这种访问方式称作 DMA(Direct Memory Access，直接内存访问)，可以绕过 CPU，不使用 CPU 访存指令访问内存。外围设备可以看到的内存地址也组成了一个地址空间。

以上三个场景均需要对一个 MR 树进行管理。QEMU 引入了 AddressSpace 数据结构(简称为 **AS**)，表示 CPU、外设或 PIO 可以访问的地址空间，定义如下。

qemu - 4.1.1/include/exec/memory.h

```
// AddressSpace: 描述 hwaddr 到 MemoryRegion 的映射
struct AddressSpace {
    char *name;                                          // AS 名称，方便调试
    MemoryRegion *root;                                  // 根 MR
    // 当前的扁平视图，在 address_space_update_topology 时作为 old 比较
    struct FlatView *current_map;

    // listener 链表，在 MR 树根 root 的拓扑结构被更改时调用，按照优先级排列
    QTAILQ_HEAD(, MemoryListener) listeners;
    QTAILQ_ENTRY(AddressSpace) address_spaces_link;     // 全局 AS 链表
};
```

AS 数据结构主要承担两个任务：一是将 MR 树(由 root 成员指向)转换成线性视图 current_map，二是在给定一个 hwaddr(GPA)时快速查找到一个 MemoryRegion，从而快速调用 MR 对应的回调函数 ops 或 MR 对应的 RB，从而找到 HVA，完成 GPA 到 HVA 的翻译。简而言之，就是完成树和线性表之间的相互转换，而线性表和树各有用途：①线性表在向 KVM 注册内存时，需要给 KVM 提供一个线性的内存区域，才可以请求 KVM 完成这段

① 参考地址：https://qemu.readthedocs.io/en/latest/devel/memory.html，包含全部 QEMU 内存接口的说明。

客户机物理内存的地址翻译以及 EPT 建立；②而树状结构方便 QEMU 调用 MR 对应的回调函数 ops，当 KVM 截获了一个 MMIO/PIO 的读写操作，且需要返回到 QEMU 时，应该在对应的 AS 中查询 MR 树，得到对应的 ops 进行模拟；也方便 QEMU 根据 GPA 找到 RAM 类型 MR 对应的 HVA，即完成 GPA 到 HVA 的转换，方便内存读写。

除了线性结构和树结构之间的转换，QEMU 还需要同步线性结构和树结构。在 AS 中，struct FlatView 类型的 current_map 是线性结构，而 MemoryRegion 类型的 root 是树状结构。由于树状结构和线性结构应该保存相同的内存拓扑结构信息，其中一个更改时，应该同步到另一个数据结构。然而，不会存在对线性结构进行修改的情况，只会对树状结构通过函数族 memory_region_* 对 MR 树进行修改。

当 MR 树被修改时，QEMU 需要重新生成线性结构，再遍历 AS 的 listeners 链表，调用每个 listener 中的函数。每当生成了一个新的线性结构 FlatView 后，都需要重新告知 KVM 需要翻译的内存区域。显而易见，每次重新告知 KVM 新的内存拓扑需要进行 ioctl 调用，这是一个系统调用，开销很大，故 QEMU 将旧的 FlatView 保存在 current_map 中，比较新旧 FlatView 得出更改部分，仅告知 KVM 要更改的部分即可。

QEMU 代码中创建了四类 AS，包括：

(1) 全局变量 address_space_memory，表示物理地址空间，其 MR 树根是大小为 UINT64_MAX 的 system_memory。

(2) address_space_io，表示 PIO 空间，其 MR 树根是大小为 65536(即 0xffff)的 system_io。

(3) 每个 CPU 都有一个名为 cpu-memory-n 的 AS，其中 n 为从 0 开始的 CPU 编号，和 address_space_memory 一样，使用了 system_memory 作为 MR 树根。

(4) 外围设备角度的 AS，例如 VGA、e1000 等模拟设备，它们的 AS 使用一个大小为 UINT64_MAX 的总线 MR 作为 MR 树根，并使用 system_memory 的别名 MR 作为 MR 树根的子 MR，即可以将内存读写转发到 system_memory 对应的区域上。

AS 数据结构提供了以下接口，供 AS 的使用者对 AS 进行创建、销毁与读写，其含义见注释。此处省略了与缓存功能相关的接口，感兴趣的读者可自行查阅源码。其中有关读写的接口说明详见 QEMU 源码树的 docs/devel/loads-stores.rst 文档。后续将围绕这些 AS 管理函数，对 AS 的树结构与线性结构的同步，以及 AS 的读写、回调进行讲解。由于一些函数复杂度较高，这里只讲解重要的函数。AS 相关操作均围绕着 hwaddr (GPA)、void * 或 uint8_t * (HVA)之间的转换进行。

qemu-4.1.1/include/exec/memory.h

```
// address_space_init: 用 MR 树根 root 初始化 AS,名称为 name
void address_space_init(AddressSpace *as, MemoryRegion *root,
                        const char *name);

// address_space_rw: 读写 AS,位置为 addr,数据位置为 buf,长度 len,其他参数 attr
MemTxResult address_space_rw(AddressSpace *as, hwaddr addr,
                        MemTxAttrs attrs, uint8_t *buf,
                        hwaddr len, bool is_write);
```

```
// address_space_write: 写入 AS,位置为 addr,数据位置为 buf,长度 len,其他参数 attr
MemTxResult address_space_write(AddressSpace * as, hwaddr addr,
                                MemTxAttrs attrs,
                                const uint8_t * buf, hwaddr len);

// address_space_map: 将 AS 的 [addr,addr + * plen) 段映射到一个 HVA 处,并返回该 HVA
void * address_space_map(AddressSpace * as, hwaddr addr,
                         hwaddr * plen, bool is_write, MemTxAttrs attrs);

// address_space_unmap: address_space_map 的逆操作
void address_space_unmap(AddressSpace * as, void * buffer, hwaddr len,
                         int is_write, hwaddr access_len);
```

在 **memory.c** 文件中,还有几个全局变量在下文出现,总结如下。

qemu-4.1.1/memory.c

```
static unsigned memory_region_transaction_depth;      // MR 事务深度,负责管理 MR 事务
static bool memory_region_update_pending;             // 是否有 MR 树的修改未同步到 FlatView
static bool ioeventfd_update_pending;                 // 是否有 ioeventfd 的更改未处理

static QTAILQ_HEAD(, MemoryListener) memory_listeners
    = QTAILQ_HEAD_INITIALIZER(memory_listeners);      // 全局的监听者链表,监听所有 AS

static QTAILQ_HEAD(, AddressSpace) address_spaces
    = QTAILQ_HEAD_INITIALIZER(address_spaces);        // 全局的 AS 链表,保存所有 AS

static GHashTable * flat_views;                       // physmr 到 FlatView 的映射,全局哈希表
```

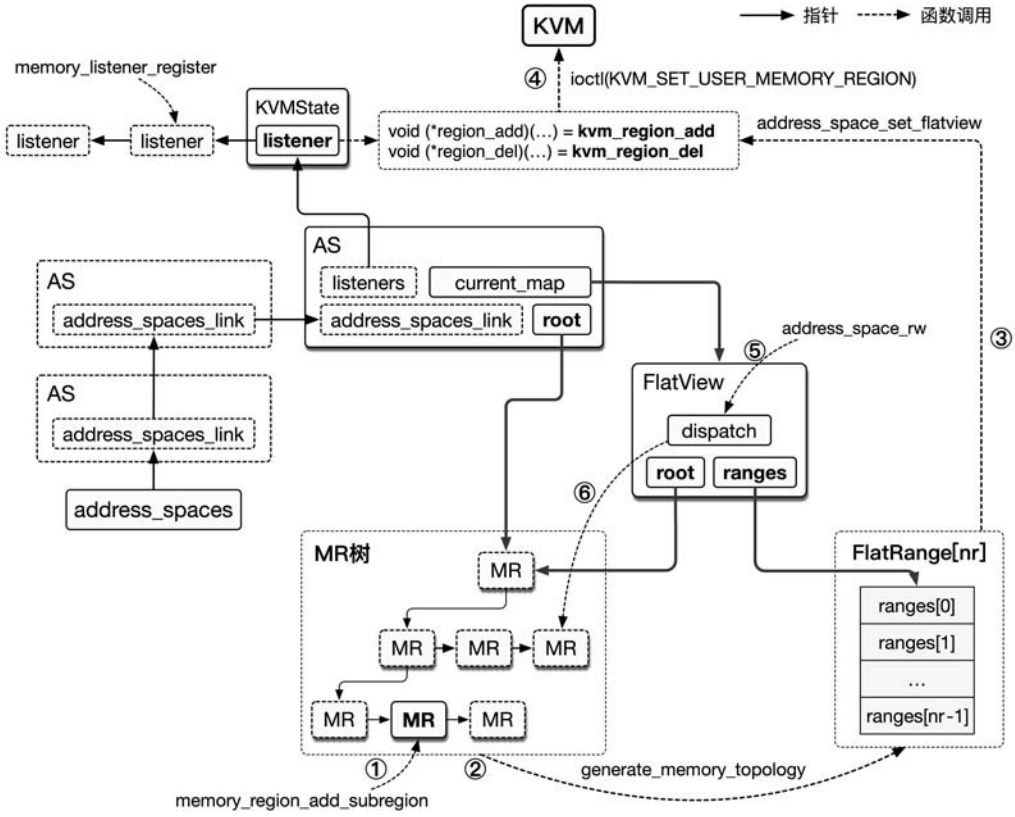
综上,AddressSpace 相关数据结构之间的关系如图 3-11 所示,下面将围绕此图的各个部分展开介绍。

4. 从树状结构到线性结构

这里介绍树状结构到线性结构的同步过程。为了将 GPA 翻译到 HPA,QEMU 通过 ioctl 系统调用,将 AS 对应的线性结构 current_map 注册到 KVM 中;当树状结构被修改时,QEMU 需要重新生成新的 current_map,并与旧的 current_map 进行比较,将更改的部分重新注册到 KVM 中。FlatView 用于管理线性结构,定义如下。

qemu-4.1.1/include/exec/memory.h

```
struct FlatView {
    unsigned ref;                                 // 引用计数
    FlatRange * ranges;                           // FlatRanges 数组
    unsigned nr;                                  // ranges 数组长度
    unsigned nr_allocated;                        // ranges 数组中有效元素的个数
    struct AddressSpaceDispatch * dispatch;       // hwaddr 地址分派器
    MemoryRegion * root;                          // 所在 AS 的 MR 树根
};
```



注：①修改 MR 树；②生成扁平视图；③通知所有 listeners；④ioctl 系统调用进入 KVM；⑤AddressSpace 读写；⑥定位到 MR 并完成读写。

图 3-11 AddressSpace 相关数据结构

每个 AS 都有一个对应的 FlatView，它保存了 AS 内存拓扑的线性结构，并且承担了 hwaddr 的分派功能，即通过 dispatch 成员将 hwaddr 映射到对应的 MR，后文介绍。FlatView 中保存了 FlatRange 数组，是 AS 对应的线性结构，FlatRange 定义如下。

qemu - 4.1.1/memory.c

```
struct FlatRange {
    MemoryRegion * mr;              // 指向对应的物理 mr
    hwaddr offset_in_region;        // 在 mr 中的偏移
    AddrRange addr;                 // 在 AS 中所占据的地址范围
    bool romd_mode;                 // Rom Device 模式
    bool readonly;                  // 只读的 FlatView
    bool nonvolatile;               // 是否是非易失性内存
    int has_coalesced_range;        // 是否存在已合并的范围
};

struct AddrRange {
    Int128 start;                   // 起始地址
```

```
    Int128 size;                    // 范围大小
};
```

如果将 FlatView 的 FlatRange 数组按顺序铺开，就得到了一个分布在 hwaddr 地址空间上的线性结构，由一段段可能互不相邻的 FlatRange 组成。何时由 MemoryRegion 树状视图生成该线性视图？经过分析，有两个时间节点：①AS 被初始化时，根据传入的 MR 树根生成线性视图 current_map；②AS 对应的 MR 树被更改时，需要重新生成线性视图 current_map。

首先分析 **AS 初始化时**如何生成 FlatView，AS 初始化函数定义如下。

qemu-4.1.1/memory.c

```
void address_space_init(AddressSpace *as, MemoryRegion *root, const char *name)
{
    as->root = root;
    as->current_map = NULL;
    QTAILQ_INIT(&as->listeners);
    QTAILQ_INSERT_TAIL(&address_spaces, as, address_spaces_link);
    address_space_update_topology(as);
}
```

该函数首先初始化各个成员，包括 current_map、root 指针，并初始化 listeners 监听者链表。全局链表 address_spaces 负责将模拟客户机硬件所使用的所有 AS 链接起来，方便遍历所有的 AS。address_update_topology 函数较为重要，该函数负责为新的 AS 生成 FlatView，定义如下。

qemu-4.1.1/memory.c

```
static void address_space_update_topology(AddressSpace *as)
{
    MemoryRegion *physmr = memory_region_get_flatview_root(as->root);

    flatviews_init() ->{
        if (flat_views)
            return;
        flat_views = g_hash_table_new_full(...);
    }

    if (!g_hash_table_lookup(flat_views, physmr)) {
        generate_memory_topology(physmr) ->{
            flatview_init(view);

            if (mr)
                render_memory_region(view, mr, ...) ->
                    flatview_insert(view, &fr);

            flatview_simplify(view);
```

```
            // 省略 dispatch 相关代码
            g_hash_table_replace(flat_views, mr, view);
            return view;
        }
    }
    address_space_set_flatview(as);
}
```

(1) 调用 memory_region_get_flatview_root 函数找到 AS 的物理 MR 根(简称 physmr,物理 MR,后文将多次用到),其目的是找到实体 MR(而非别名 MR)的树根。这样可以减少 FlatView 的个数,使得拥有相同的实体 MR 的 AS 共用一个 FlatView,减少内存开销,详见注释①。由此可知,AS 的 FlatView 与 physmr 绑定,而非与根 MR 绑定。

(2) 调用 flatviews_init 函数初始化全局变量 flat_views,它是一个全局的哈希表,负责将 MR 映射到 FlatView。在首次调用 flatviews_init 函数时被设置为新的空哈希表。

(3) 调用 generate_memory_topology 函数,生成 physmr 对应的 FlatView。这是将树状结构转换为线性结构的核心函数,但由于其复杂程度较高,此处不进行详细分析。它首先初始化 view,然后调用 render_memory_region 函数生成 physmr 对应的 view,再调用 flatview_simplify 函数简化 view。接下来的代码将根据生成的 view 填充地址分派器 dispatch。最终,physmr 到 view 的映射被存储在全局哈希表 flat_views 中。

至此,QEMU 已经将树状结构转换为线性结构,接下来是告知所有**监听者**新的线性结构。在这里,只有一个 KVM 监听器,代码如下。

qemu-4.1.1/include/sysemu/kvm_int.h

```
typedef struct KVMSlot
{
    hwaddr start_addr;                          // 起始地址
    ram_addr_t memory_size;                     // Slot 大小
    void * ram;                                 // 宿主机虚拟地址,即 HVA
} KVMSlot;

typedef struct KVMMemoryListener {
    MemoryListener listener;                    // 指向通用 MemoryListener
    KVMSlot * slots;                            // KVMSlot 数组
} KVMMemoryListener;
```

KVMMemoryListener 中的 KVMSlot 是 QEMU 中 KVM 的 kvm_memory_slot 对应的数据结构,负责向 KVM 注册内存；listener 是通用监听器,定义如下。

① 地址：https://patchwork.kernel.org/project/qemu-devel/patch/20170921085110.25598-10-aik@ozlabs.ru/,说明了引入 physmr 的原因。

qemu - 4.1.1/include/exec/memory.h

```
struct MemoryListener {
    void ( * region_add)(MemoryListener * listener, MemoryRegionSection * section);
                                                    // 添加函数
    void ( * region_del)(MemoryListener * listener, MemoryRegionSection * section);
    unsigned priority;                              // 优先级
    AddressSpace * address_space;
    QTAILQ_ENTRY(MemoryListener) link;              // listener 链表节点
};
```

通用监听器 MemoryListener 是一个函数指针的集合，并有一个 priority 成员表示其优先级，所有的 listener 在注册时按照优先级顺序连接到 AS 的监听者链表上。其中 KVM 监听器注册的代码如下。

qemu - 4.1.1/accel/kvm/kvm - all.c

```
void kvm_memory_listener_register(...) {
    kml -> listener.region_add = kvm_region_add;
    kml -> listener.region_del = kvm_region_del;
    kml -> listener.priority = 10;
    memory_listener_register(&kml -> listener, as);
}

kvm_init ->
    kvm_memory_listener_register(&s -> memory_listener,
                                 &address_space_memory, 0);
kvm_region_add ->
kvm_set_phys_mem(kml, section, true) ->
kvm_set_user_memory_region(kml, mem, true) ->
kvm_vm_ioctl(s, KVM_SET_USER_MEMORY_REGION, &mem) ->
ioctl(s -> vmfd, KVM_SET_USER_MEMORY_REGION, &mem)     // 进入内核
```

在 KVM 监听者中，region_add 函数指针指向 kvm_region_add 函数，最终调用 ioctl 函数完成内存的注册。下面继续介绍 AS 的初始化流程。

(4) 最后回到 AS 的初始化函数 address_space_init，接步骤(3)。它继续调用 address_space_set_flatview 函数，将新生成的 physmr 对应的 view 告知所有监听者，并且将 AS 的 current_map 更新到新生成的 view。此处调用的是 KVM 监听器的回调函数，从 address_space_set_flatview 函数讲起，代码如下。

qemu - 4.1.1/memory.c

```
static void address_space_set_flatview(AddressSpace * as)
{
    FlatView * old_view = address_space_to_flatview(as);
    MemoryRegion * physmr = memory_region_get_flatview_root(as -> root);
    FlatView * new_view = g_hash_table_lookup(flat_views, physmr);
```

```
    if (old_view == new_view)
        return;

    if (!QTAILQ_EMPTY(&as->listeners)) {
        ...
        address_space_update_topology_pass(as, old_view2, new_view, false);
        address_space_update_topology_pass(as, old_view2, new_view, true);
    }
    atomic_rcu_set(&as->current_map, new_view);
}
```

address_space_to_flatview 函数首先找到旧的 current_map，作为 old_view，该情况下为 NULL；再通过 physmr 在全局哈希表 flat_views 中查找新生成的 physmr 对应的 view，作为 new_view，将新旧 view 比较。在 AS 初始化时，如果新旧 view 不相同，则会将新旧 view 传给 address_space_update_topology_pass 函数，从而告知所有的监听者线性结构的变化，最后将 current_map 更新为 new_view。线性结构变化的单位是 MemoryRegionSection，定义如下。

qemu-4.1.1/include/exec/memory.h

```
struct MemoryRegionSection {
    Int128 size;                            // section 大小
    MemoryRegion *mr;                       // 对应的 MR
    FlatView *fv;                           // 对应的 FlatView
    hwaddr offset_within_region;            // 在 MR 中的偏移
    hwaddr offset_within_address_space;     // 在 AS 中的偏移
    bool readonly;                          // 只读的 FlatView
    bool nonvolatile;                       // 是否是非易失性内存
};
```

address_space_update_topology_pass 函数首先对比新旧 FlatView，得出新旧 FlatView 之间的差别，用 FlatRange 的形式保存。对此 FlatRange 调用 MEMORY_LISTENER_UPDATE_REGION 函数将 FlatView 转化为 MemoryRegionSection，准备好调用所有 listener 的 region_add 函数。最终遍历 AS 的 listeners 链表，使用 MemoryRegionSection 调用所有 listener 的 region_add 函数。此处只关注 KVM 的 kvm_region_add 函数，这在前文介绍监听器数据结构时已经介绍过。具体调用链如下。

qemu-4.1.1/memory.c

```
address_space_update_topology_pass ->{
    while (iold < old_view->nr || inew < new_view->nr) {
        // 省略比较新旧 FlatView 的逻辑
          MEMORY_LISTENER_UPDATE_REGION(frold, as, Reverse, region_add);
        } // 省略其他存在更改的情况
    }
}
```

```
#define MEMORY_LISTENER_UPDATE_REGION(fr, as, dir, callback, _args...)          \
    do {                                                                        \
        MemoryRegionSection mrs = section_from_flat_range(fr,                   \
                address_space_to_flatview(as));                                 \
        MEMORY_LISTENER_CALL(as, callback, dir, &mrs, ##_args);                 \
    } while(0)

#define MEMORY_LISTENER_CALL(_as, _callback, _direction, _section, _args...)    \
    do {                                                                        \
        MemoryListener *_listener;                                              \
        // 省略 switch                                                          \
        QTAILQ_FOREACH(listener, &(_as)->listeners, link_as) {                  \
            if (_listener->_callback) {                                         \
                _listener->_callback(listener, _section, ##_args);              \
            }                                                                   \
        }                                                                       \
    } while (0)
```

至此，QEMU 已经完成了 AS 初始化工作。AS 初始化时涉及的调用链如图 3-12 所示。可以看到，初始化一个 AS 时，首先调用 generate_memory_topology 函数生成其 physmr 根对应的 FlatView，再调用函数 address_space_set_flatview→address_space_update_topology_pass→kvm_region_add 通知 KVM 模块：线性视图已经更改，需要重新向 KVM 使用 ioctl 函数注册内存，最终调用 ioctl 函数进入内核态的 KVM 模块中。

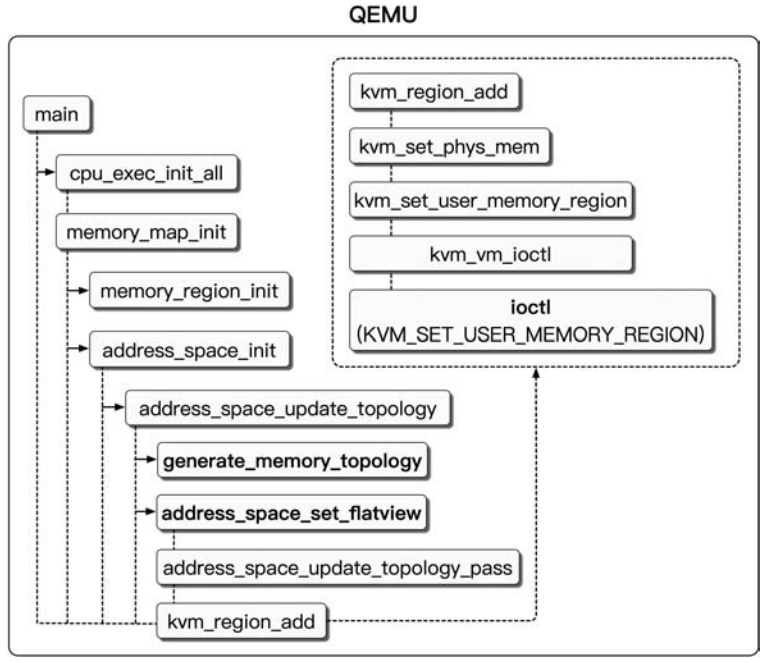


图 3-12　AS 创建时树状结构到线性结构的调用链

简而言之，重要的是 generate_memory_topology 函数，负责生成 MR 树对应的 FlatView；以及 address_space_set_flatview 函数，负责将 FlatView 的更改通过 address_space_update_topology_pass 函数告知所有 listener，其中包括 KVMMemoryListener。

AS 的初始化只是一个需要同步树状结构与线性结构的情况。在 AS 初始化之后，MR 树被更新时也需要同步到 FlatView，并通知 KVM。QEMU 提供的多个操作 MR 的接口会更新 MR 树，如 memory_region_add_subregion 函数，QEMU 都会使用 MR 事务机制完成 FlatView 的同步以及 KVM 监听器的通知，其大致调用链代码如下。

qemu-4.1.1/memory.c

```
memory_region_*() ->{                          // 更新 MR 的一类函数
    memory_region_transaction_begin ->         // 事务开始
        { ++memory_region_transaction_depth; }
    ... // 更新 MR 树中的 MR
    memory_region_transaction_commit ->{       // 事务提交
        --memory_region_transaction_depth;
        if (!memory_region_transaction_depth) {
            if (memory_region_update_pending) {
                flatviews_reset() ->{
                    QTAILQ_FOREACH(as, &address_spaces,address_spaces_link) {
                        generate_memory_topology(physmr);
                    }
                }
                memory_region_update_pending = false;
                QTAILQ_FOREACH(as, &address_spaces, address_spaces_link) {
                    address_space_set_flatview(as);
                }
            } else if (ioeventfd_update_pending) {
                // 省略 ioeventfd 相关处理
            }
        }
    }
}
```

可以看到，每次修改 MR 树都在 flatviews_reset 函数中重新生成了对应的 FlatView，并且调用 address_space_set_flatview 函数将新的 FlatView 注册到 KVM 中。简化的调用链如图 3-13 所示，类似于 AS 创建之后的调用链。

至此，QEMU 完成了树状结构到线性结构的同步，并将线性结构注册到 KVM 中。QEMU 进行了树状结构到线性结构的转化，将较为复杂的“策略”转换成一种简单的形式，可以调用 KVM 提供的“机制”完成 QEMU/KVM 的协同工作。此时，当客户机访问一个“虚拟”物理地址 GPA 时，如果该 GPA 不是用于 MMIO，那么 MMU 都会查询 KVM 所维护的 EPT 得到其对应的“真实”物理地址，客户机访问这部分“虚拟”物理内存将不会引起 VM-Exit，从而完成高效的内存虚拟化。对于 MMIO 区域的 GPA，QEMU 是如何与 KVM 协作的呢？

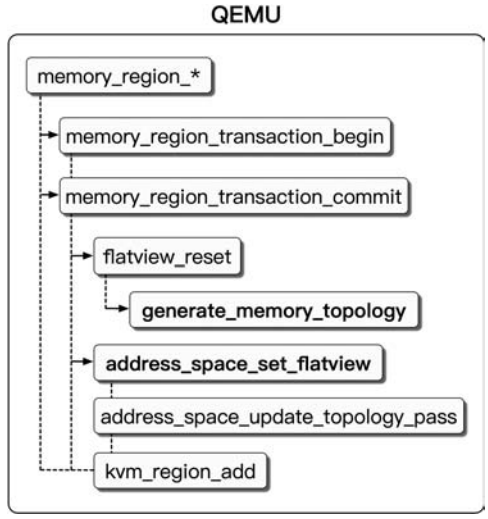


图 3-13　更改 MemoryRegion 树时树状结构到线性结构的调用链

5. 客户机物理内存地址的分派

对于实现 MMIO 的 MR，在 KVMMemoryListener 对它进行 KVM 内存注册时，注册函数 kvm_region_add 将其识别为 MMIO 对应的“虚拟”物理内存区，没有对应的宿主机虚拟地址，就不会将其提交到 KVM 中。如果客户机访问了这段内存，KVM 会识别这是 MMIO 对应的内存区，最终返回到 QEMU 的 ioctl(KVM_RUN)系统调用之后。对于 PIO 的处理是类似的，也会退出到 QEMU 的 ioctl(KVM_RUN)系统调用之后。调用代码如下。

qemu-4.1.1/accel/kvm/kvm-all.c

```
qemu_kvm_start_vcpu -> qemu_kvm_cpu_thread_fn -> kvm_cpu_exec -> {
    do {
        run_ret = kvm_vcpu_ioctl(cpu, KVM_RUN, 0);        // 进入内核
        switch (run->exit_reason) {                        // 内核返回的退出原因
        case KVM_EXIT_IO:
            kvm_handle_io() ->
                address_space_rw(&address_space_io, run-> ...)
        case KVM_EXIT_MMIO:
            address_space_rw(&address_space_memory, run-> ...)
        }
    } while (ret == 0);
}
```

可以看到，在退出到 QEMU 之后，需要对 QEMU 模拟 MMIO/PIO 所使用的 AS 数据结构进行读写，使用的是前文介绍的 AS 读写函数 address_space_rw。所谓**客户机物理内存地址的分派**是指，将对 AS 读写的**地址**分派到对应的 MemoryRegion 上，调用 MR 所包含的处理函数进行 MMIO/PIO 模拟。

事实上，如果给定一个 hwaddr，可以在 MR 树中搜索其对应的 MR，但这样做无疑是很低效的。为此，QEMU 引入了一个类似于页表的结构完成地址转换，即 struct AddressSpaceDispatch，其复杂程度较高，不进行深入分析。数据结构关系如下。

qemu-4.1.1/include/exec/memory.h

```
struct AddressSpace {
    struct FlatView * current_map;
}
struct FlatView {
    struct AddressSpaceDispatch * dispatch;
}
```

AS 的线性视图（扁平视图）中保存了 struct AddressSpaceDispatch 的指针，定义如下。

qemu-4.1.1/exec.c

```
typedef struct PhysPageEntry PhysPageEntry;
struct PhysPageEntry {
    uint32_t skip : 6;
    uint32_t ptr : 26;
};
typedef PhysPageEntry Node[P_L2_SIZE];
typedef struct PhysPageMap {
    unsigned sections_nb, sections_nb_alloc;
    unsigned nodes_nb, nodes_nb_alloc;
    Node * nodes;
    MemoryRegionSection * sections;
} PhysPageMap;

struct AddressSpaceDispatch {
    MemoryRegionSection * mru_section;          // 最近访问的 section
    PhysPageEntry phys_map;                     // 页表指针
    PhysPageMap map;                            // 页表
};
```

每个 AS 有一个独立的 AddressSpaceDispatch，作为 AS 内存地址分派的页表，其叶子结点的页表项指向 MemoryRegionSection，查询该页表的结果是 MemoryRegionSection。在 AddressSpaceDispatch 中，mru_section 保存了最近一次的查询结果，作用类似于 TLB；map 是一个多级页表，即能够快速查找，又不会占用过多内存。phys_map 起到了 CR3 的作用，PhyPageMap 中的 nodes 表示页表的中间节点，sections 等同于物理页的作用。

由于 AddressSpaceDispatch 实现较为复杂，这里只关注该数据结构提供的接口。正如前文提到的，AS 对应的 dispatch 在生成线性视图时初始化，并填入 FlatView 的 dispatch 字段，即在前文提到的核心函数 generate_memory_topology 中初始化，代码如下。

qemu-4.1.1/memory.c

```
// FlatRange 转换为 MemoryRegionSection
```

```
static inline MemoryRegionSection
section_from_flat_range(FlatRange * fr, FlatView * fv) {
    return (MemoryRegionSection) {
        .mr = fr->mr,
        .fv = fv,
        .offset_within_region = fr->offset_in_region,
        .size = fr->addr.size,
        .offset_within_address_space = int128_get64(fr->addr.start),
        .readonly = fr->readonly,
        .nonvolatile = fr->nonvolatile,
    };
}
static FlatView * generate_memory_topology(MemoryRegion * mr) {
    // 省略生成 FlatRange 数组的过程
    view->dispatch = address_space_dispatch_new(view);
    for (i = 0; i < view->nr; i++) {
        MemoryRegionSection mrs =
            section_from_flat_range(&view->ranges[i], view);
        flatview_add_to_dispatch(view, &mrs);
    }
    address_space_dispatch_compact(view->dispatch);
}
```

在生成 AS 的 FlatRange 数组之后，QEMU 将 FlatRange 数组中每个元素转换成其对应的 MemoryRegionSection，并调用 flatview_add_to_dispatch 函数填入页表 AddressSpaceDispatch 中。这意味着，只要生成了 FlatRange，QEMU 就可以使用 AddressSpaceDispatch 查找一个 AS 中 hwaddr(GPA)所在的 MR，从而得出 GPA 对应的 HVA。

address_space_rw 函数使用页表 AddressSpaceDispatch 完成地址转换，定义如下。

qemu-4.1.1/exec.c

```
MemTxResult address_space_rw ->{
    address_space_write(as, addr, attrs, buf, len) /
    address_space_read_full(as, addr, attrs, buf, len) ->{
        if (len > 0) {
            fv = address_space_to_flatview(as) ->
                { return atomic_rcu_read(&as->current_map); }
            result = flatview_write(fv, addr, attrs, buf, len);
        }
        return result;
    }
}
```

这里出现了一个 MemTxResult 类型，QEMU 将对 AS 的读写视为一次 MR 事务，其返回结果 MemTxResult 是一个 uint32_t 类型的变量，定义在 include/exec/memattrs.h 中，可以取 MEMTX_OK、MEMTX_ERROR 等值。对 AS 进行读写时，首先原子地读取 AS 的 current_map，即当前的线性结构；再调用 flatview_read/write 函数对线性结构 fv 进行读

写。此处用 flatview_read 函数作为例子进行说明，其定义如下。

qemu-4.1.1/exec.c

```
flatview_read(FlatView fv, hwaddr addr) -> {
    mr = flatview_translate(fv, addr);
    return flatview_read_continue(mr, addr) -> {
        for (;;) {
            if (!memory_access_is_direct(mr, false)) { // I/O 区域的读写
                memory_region_dispatch_read -> {
                    tmp = mr->ops->read(mr->opaque, addr, size);
                }
            } else { // RAM 区域的读写
                ptr = qemu_ram_ptr_length(mr->ram_block, addr1, &l, false);
                    memcpy(buf, ptr, l);
            }
            ...
            mr = flatview_translate(fv, addr, &addr1, &l, false, attrs);
        }
    };
}
```

可以看到，flatview_read 函数不断调用 flatview_translate 函数，通过 FlatView 内部的页表 AddressSpaceDispatch 得到一个 hwaddr 地址对应的 MemoryRegion，再进行 MemoryRegion 对应的模拟。对于 I/O 类型的 MR，最终调用 ops-> read 函数完成读取的模拟；对于 RAM 类型的 MR，则找到 hwaddr addr 对应的 HVA，记录在 ptr 指针中，调用 memcpy 完成读取。客户机物理内存地址分派的调用链如图 3-14 所示。

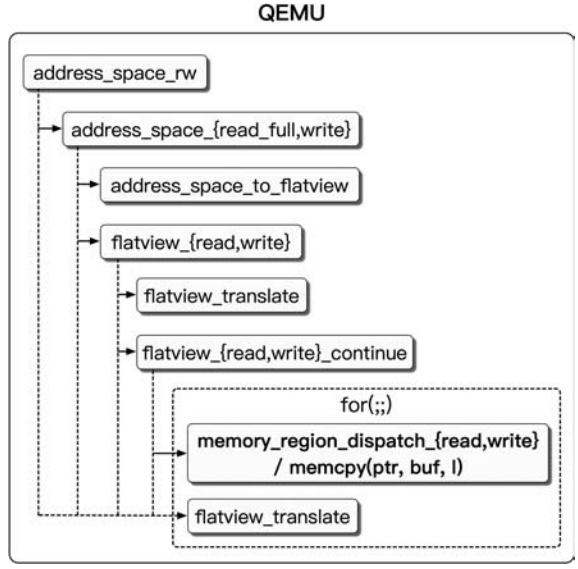


图 3-14 客户机物理内存地址分派

至此，已经介绍了 QEMU 中内存虚拟化相关的大部分数据结构及其操作接口之间的关系。总结如下：①AddressSpace 是顶层数据结构，将 MemoryRegion 树和 FlatView 线性结构组织起来，形成一个可供读写的地址空间。②MemoryRegion 中的容器类型和别名类型分别模拟了真实系统中的总线和内存控制器，将 I/O 类型的 MR 和 RAM 类型的 MR 通过树的形式组织起来；I/O 类型的 MR 提供了一组 ops 回调函数，供 QEMU 实现物理硬件的模拟，而 RAM 类型的 MR 对应宿主机上的一段虚拟地址 HVA，作为客户机的“虚拟”物理地址，QEMU 最终将这段地址注册到 KVM 中完成 GPA 到 HPA 的翻译。③FlatView 保存了 MemoryRegion 树对应的线性结构 FlatRange，供 QEMU 将其转换为 MemoryRegionSection 注册到 KVM 中；还保存了地址分派器 AddressSpaceDispatch，负责将 GPA 翻译为 HVA。下一节将展示运行过程中这些数据结构的组织形式，使读者有更直观的认识。

3.3.2　实验：打印 MemoryRegion 树

QEMU 为了模拟 MMIO 以及物理设备的行为，形成了一套复杂的数据结构，但这些只是静态的代码。本节将 QEMU 代码运行起来，在动态过程中打印出 MemoryRegion 树，更形象地展示数据结构之间的关系。

实验使用从源代码编译的 QEMU v4.1.1，以及事先准备好的客户机磁盘镜像作为 QEMU 的-hda 参数传递给 QEMU。首先，使用如下命令进入 QEMU 监视器。

Physical Machine Terminal 1

```
sudo ./qemu-4.1.1/x86_64-softmmu/qemu-system-x86_64
     -m 4096 -smp 2 -cpu host
     --enable-kvm -monitor stdio
     -numa node,cpus=0 -numa node,cpus=1
QEMU 4.1.1 monitor - type 'help' for more information
(qemu) VNC server running on 127.0.0.1:5900
(qemu)
```

启动命令的含义为：将 QEMU 管理器的输入输出重定向到字符设备 stdio(-monitor stdio)，即此处的命令行；此命令启动了 2 个 vCPU(-smp 2)，使用 NUMA(Non-Uniform Memory Access，非统一内存访问)架构，分为两个 NUMA 节点(-numa node)，分配 4GB 的“虚拟”物理内存(-m 4096)；开启 KVM 支持，并使用与宿主机一样的 CPU 型号(-cpu host --enable-kvm)。接下来，使用命令 info mtree 打印此客户机的 MemoryRegion 树，在输出中，QEMU 用不同宽度的缩进表示不同树的深度，打印如下。

Physical Machine Terminal 1

```
(qemu) info mtree
address-space: memory
  0000000000000000-ffffffffffffffff (prio 0, i/o): system
    0000000000000000-00000000bfffffff (prio 0, i/o): alias ram-below-4g @pc.ram
0000000000000000-00000000bfffffff
```

```
    0000000000000000-ffffffffffffffff (prio -1, i/o): pci
      00000000000a0000-00000000000bffff (prio 1, i/o): vga-lowmem
      00000000000c0000-00000000000dffff (prio 1, rom): pc.rom
      00000000000e0000-00000000000fffff (prio 1, i/o): alias isa-bios @ pc.bios
0000000000020000-000000000003ffff
      00000000fd000000-00000000fdffffff (prio 1, ram): vga.vram
      00000000febc0000-00000000febdffff (prio 1, i/o): e1000-mmio
      00000000febf0000-00000000febf0fff (prio 1, i/o): vga.mmio
        00000000febf0000-00000000febf00ff (prio 0, i/o): edid
      00000000fffc0000-00000000ffffffff (prio 0, rom): pc.bios
    00000000fec00000-00000000fec00fff (prio 0, i/o): kvm-ioapic
    00000000fed00000-00000000fed003ff (prio 0, i/o): hpet
    00000000fee00000-00000000feefffff (prio 4096, i/o): kvm-apic-msi
    0000000100000000-000000013fffffff (prio 0, i/o): alias ram-above-4g @ pc.ram
00000000c0000000-00000000ffffffff

address-space: I/O
  0000000000000000-000000000000ffff (prio 0, i/o): io
    0000000000000000-0000000000000007 (prio 0, i/o): dma-chan
    0000000000000008-000000000000000f (prio 0, i/o): dma-cont
    0000000000000064-0000000000000064 (prio 0, i/o): i8042-cmd
    0000000000000070-0000000000000071 (prio 0, i/o): rtc
      0000000000000070-0000000000000070 (prio 0, i/o): rtc-index
    000000000000007e-000000000000007f (prio 0, i/o): kvmvapic
    0000000000000080-0000000000000080 (prio 0, i/o): ioport80
    0000000000000081-0000000000000083 (prio 0, i/o): dma-page
    0000000000000087-0000000000000087 (prio 0, i/o): dma-page
    0000000000000089-000000000000008b (prio 0, i/o): dma-page
    000000000000008f-000000000000008f (prio 0, i/o): dma-page
    0000000000000092-0000000000000092 (prio 0, i/o): port92
    00000000000000a0-00000000000000a1 (prio 0, i/o): kvm-pic

address-space: cpu-memory-0
  // 与 address-space: memory 相同

address-space: cpu-memory-1
  // 与 address-space: memory 相同

address-space: i440FX
  0000000000000000-ffffffffffffffff (prio 0, i/o): bus master container

address-space: PIIX3
  0000000000000000-ffffffffffffffff (prio 0, i/o): bus master container

address-space: VGA
  0000000000000000-ffffffffffffffff (prio 0, i/o): bus master container

address-space: e1000
  0000000000000000-ffffffffffffffff (prio 0, i/o): bus master container
    0000000000000000-ffffffffffffffff (prio 0, i/o): alias bus master @system
```

```
    0000000000000000 - ffffffffffffffff

address - space: piix3 - ide
  0000000000000000 - ffffffffffffffff (prio 0, i/o): bus master container

memory - region: pc.ram
  0000000000000000 - 00000000ffffffff (prio 0, ram): pc.ram

memory - region: pc.bios
  00000000fffc0000 - 00000000ffffffff (prio 0, rom): pc.bios

memory - region: pci
  // 与 address - space: memory 中对应的部分相同

memory - region: system
  // 与 address - space: memory 中对应的部分相同
```

此处省略了被标为［disabled］的 MR，以及一些陌生的 MR。可以看到，整个虚拟机有 address_space_memory 作为物理内存空间的 AS，有 address_space_io 作为 PIO 端口映射空间的 AS。由于本实验启动了 2 个 vCPU，所以这里打印出了两个 CPU 的 AS，即 cpu-memory-0/1。其他的 AS 包括从设备角度可以观察到的 AS，如 e1000、VGA 等设备的 AS。每个 AS 下显示了 AS 的 MR 树，其中非别名类型的 MR 只能打印出一条较短的记录，包含其地址范围。如 address_space_memory 的 MR 树根 system_memory，其地址范围是 0x0000000000000000～0xffffffffffffffff，即 0～UINT64_MAX；而别名 MR 会被明确标识为 alias，并追加上其 alias 指针指向的原 MR。有关 info mtree 命令的实现函数，请查阅 QEMU 源码树 memory.c 文件的 mtree_info→mtree_print_mr 函数。

为了与源码相对应，继续在 QEMU 源码中寻找这些 AS 和 MR 被创建的位置，具体方法多种多样。一种直接的方法是在源码中搜索相关的创建函数，如 address_space_init、memory_region_init，更严谨的方法是通过 GDB 打断点的方式寻找。首先，在 QEMU 的 main 函数中，cpu_exec_init_all 函数初始化了主要的 AS 以及 MR 树根，代码如下。

qemu - 4.1.1/exec.c

```
// 全局变量、静态变量
RAMList ram_list = { .blocks = QLIST_HEAD_INITIALIZER(ram_list.blocks) };
static MemoryRegion * system_memory, * system_io;
AddressSpace address_space_io, address_space_memory;
MemoryRegion io_mem_rom, io_mem_notdirty;
main() ->
cpu_exec_init_all() -> {
    io_mem_init() -> {
        memory_region_init_io(&io_mem_rom);
        ...
    }
    memory_map_init() -> {
        memory_region_init(system_memory, NULL, "system", UINT64_MAX);
```

```
        address_space_init(&address_space_memory, system_memory, "memory");
        memory_region_init_io(system_io, NULL, &unassigned_io_ops, NULL, "io", 65536);
        address_space_init(&address_space_io, system_io, "I/O");
    }
}
```

在这里，QEMU 初始化了 system_memory/system_io 等静态变量，作为两个全局 AS 变量 address_space_memory/address_space_io 的 MR 树根。这里初始化了与体系结构无关的 AS，下面进入 i386 的模拟部分中与初始化架构相关的部分。在不同类型的 PC_MACHINE 的定义函数中，也会初始化 AS/MR 等数据结构，以 pc_init1 函数为例。

qemu-4.1.1/hw/i386/pc_piix.c

```
// PC 硬件初始化函数
pc_init1()                                                          // pc_piix.c
{
    if (pcmc->pci_enabled) {                                        // 初始化 PCI MR
        memory_region_init(pci_memory, NULL, "pci", UINT64_MAX);
        rom_memory = pci_memory;
    }

    pc_memory_init(pcms, system_memory, rom_memory, &ram_memory)    //pc.c
    {
        memory_region_allocate_system_memory(ram, NULL, "pc.ram",
                                machine->ram_size)                  // numa.c
        {
            if (nb_numa_nodes == 0 || !have_memdevs) {              //模拟非 NUMA
                allocate_system_memory_nonnuma(ram) ->
                    memory_region_init_ram_nomigrate(ram) ->
                        new_block->host = qemu_ram_mmap() -> mmap()
            } else {
                // 省略模拟 NUMA 架构的代码
            }
        } // memory_region_allocate_system_memory

        memory_region_init_alias(ram_below_4g, NULL, "ram-below-4g", ram,
                        ...);
        memory_region_add_subregion(system_memory, 0, ram_below_4g);

        if (pcms->above_4g_mem_size > 0) {
          memory_region_init_alias(ram_above_4g, NULL, "ram-above-4g", ram,
                          ...);
          memory_region_add_subregion(system_memory, 0x100000000ULL,
                          ram_above_4g);
        }
        memory_region_init_ram(option_rom_mr, NULL, "pc.rom", PC_ROM_SIZE,
                          &error_fatal);
    } // pc_memory_init
} // pc_init1
```

可以看到，pc_init1 函数首先初始化了 PCI MR，继续调用 pc_memory_init 函数，初始化了真实的全局物理内存 pc.ram MR，并将其分为两个别名 MR，即 ram_below_4g/ram_above_4g，并作为子 MR 加入了 system_memory。在解析 QEMU 参数时，QEMU 读取到-m 参数后的数字，并将其保存在 machine-> ram_size 中，作为初始化 pc.ram MR 的大小，即物理内存的大小。在分配全局 pc.ram MR 时，QEMU 将 NUMA 和非 NUMA 的情况分类。

非 NUMA 的情况下，直接分配一个 RAM 类型的实体 MR 即可；而 NUMA 情况下，需要调用 host_memory_backend_get_memory 函数得到每个 NUMA 节点对应的 MR，并作为子 MR 加入 pc.ram MR 中。这与之前 info mtree 打印出来的 MR 树相符合。

3.3.3　KVM 内存数据结构

相比于“策略”的实现，“机制”的实现往往更加简单。QEMU 内存虚拟化需要完成的功能较多，包括宿主机虚拟内存的分配、MMIO 的模拟、HVA 到 GPA 的翻译等，而 KVM 内存虚拟化只需要维护好 EPT 页表，并与硬件配合即可。同时，由于 KVM 是 Linux 内核中的一个内核模块，它可以重用 Linux 内核的内存管理接口，降低了实现的难度。

本节不讨论由于性能不佳而不常采用的影子页表的实现，只讨论 Intel x86 架构下的扩展页表 EPT 的维护。在 Linux 源码树的 **Documentation** 目录下，描述内核代码的 **Documentation/virtual/kvm/mmu.txt** 文档有对 KVM 内存管理模块较为全面规范的描述，但较难理解。下文将抽取主线，使叙述更易懂。KVM 中相关的数据结构相比于 QEMU 更简洁，如图 3-15 所示。

1. 接收 QEMU 的内存注册

首先，QEMU 应该给 KVM 注册需要其做地址翻译的“虚拟”物理内存，否则 KVM 所维护的 EPT 页表将无用武之地，因此从 KVM 接收 QEMU 的内存注册开始讲起。联系 3.3.1 节，当 MemoryRegion 树被更改后，都会通知所有的 listener，其中包括 KVM 的监听器 KVMMemoryListener，调用 ioctl 完成 KVM 内存注册。注册的基本单位是如下数据结构，包含 GPA、HVA、该段内存的大小等字段，与 QEMU 中的线性视图相类似。

linux-4.19.0/include/uapi/linux/kvm.h

```
#define KVM_MEM_LOG_DIRTY_PAGES (1UL << 0)      // 需要记录脏页
#define KVM_MEM_READONLY (1UL << 1)             // 只读

/* for KVM_SET_USER_MEMORY_REGION */
struct kvm_userspace_memory_region {
    __u32 slot;                                 // 保存 ID 号和 AS 的 ID 号
    __u32 flags;                                // 标志，有效的标志只有 0 和 1，见上面宏定义
    __u64 guest_phys_addr;                      // GPA
    __u64 memory_size;                          // 该段内存大小
    __u64 userspace_addr;                       // HVA
};
```

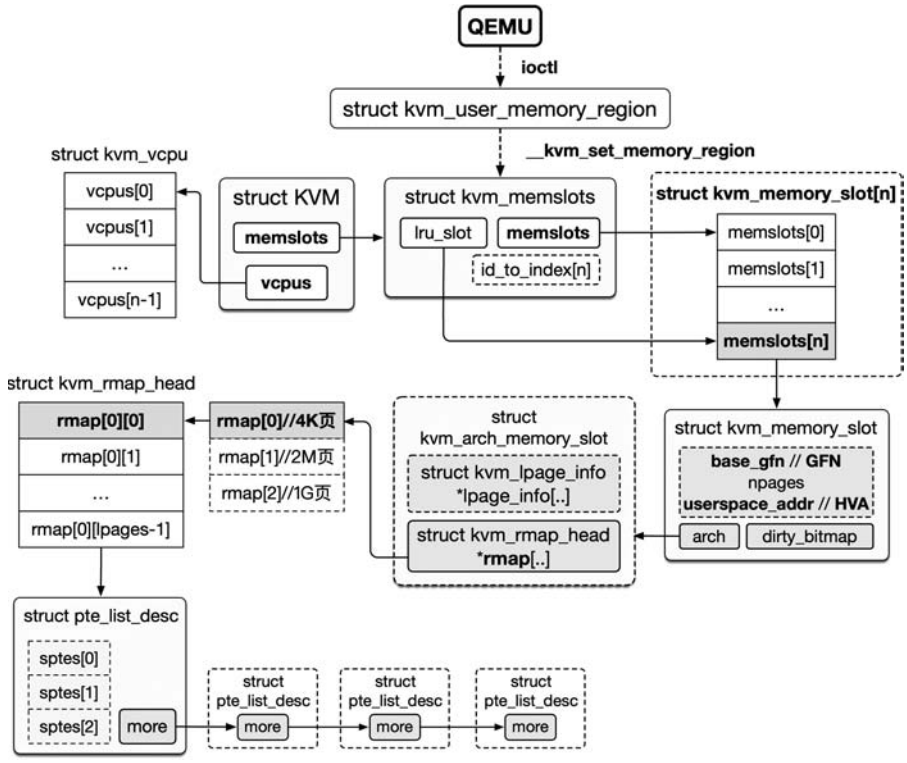


图 3-15 KVM 内存虚拟化数据结构

QEMU 进行 ioctl 系统调用后进入内核的 kvm_vm_ioctl 函数，并根据 ioctl 的参数调用相应的处理函数。此时 ioctl 参数是如下 KVM_SET_USER_MEMORY_REGION。

linux-4.19.0/virt/kvm/kvm_main.c

```
static long kvm_vm_ioctl(...unsigned int ioctl, unsigned long arg) {
    switch (ioctl) {
    case KVM_SET_USER_MEMORY_REGION:  {
        struct kvm_userspace_memory_region kvm_userspace_mem;
        copy_from_user(&kvm_userspace_mem, argp, sizeof(kvm_userspace_mem));

        kvm_vm_ioctl_set_memory_region(kvm, &kvm_userspace_mem) ->
          kvm_set_memory_region(kvm, mem) ->
            __kvm_set_memory_region(kvm, mem)
        break;
    }
}
```

由于 QEMU 中的 kvm_userspace_memory_region 处在用户态，因此内核态的 KVM 需要使用 copy_from_user 函数将其中的数据复制到内核态。最终进入__kvm_set_memory_region 函数，将 kvm_userspace_memory_region 注册到 KVM 中，而 KVM 中保存

客户机内存线性视图的结构是 kvm_memory_region，用户态 QEMU 传来的数据需要转化为该数据结构进行保存，定义如下。

linux - 4.19.0/include/linux/kvm_host.h

```
struct kvm_memory_slot {
    gfn_t base_gfn;                     // 起始 GFN
    unsigned long npages;               // slot 的大小，单位是 4KB 的页
    unsigned long * dirty_bitmap;       // 脏页位图
    struct kvm_arch_memory_slot arch;   // 架构相关信息
    unsigned long userspace_addr;       // 起始 HVA
    u32 flags;                          // 和 user 态 memslot 的标志位相同
    short id;
};
struct kvm_memslots {
    u64 generation;
    // kvm_memory_slot 数组，按照 base_gfn 从大到小排序，形成 gfn_t 的地址空间
    struct kvm_memory_slot memslots[KVM_MEM_SLOTS_NUM];
    // 用 kvm_memory_slot.id 查询在 memslots 数组中的 index
    short id_to_index[KVM_MEM_SLOTS_NUM];
    atomic_t lru_slot;                  // 最近使用的 slot 在 memslots 中的索引
    int used_slots;                     // memslots 中有效元素的个数
};
struct kvm {
    struct kvm_memslots __ rcu * memslots[KVM_ADDRESS_SPACE_NUM]; // 两个 AS
}
enum kvm_mr_change {
    KVM_MR_CREATE, KVM_MR_DELETE, KVM_MR_MOVE, KVM_MR_FLAGS_ONLY,
};
```

可以看到，每个 KVM 虚拟机对应的 struct kvm 结构体都保存了一个 memslots，其中保存了 kvm_memory_slot 的数组。这是 KVM 中唯一保存**客户机物理页**信息的位置，与 QEMU 不同，KVM 仅有一个客户机物理内存的线性视图，保存了所有客户机物理内存的相关信息。在进入 KVM 的内存注册 ioctl 后，__ kvm_set_memory_region 函数将使用用户态传来的结构体 kvm_userspace_memory_region 更新 kvm 的 memslots，更新类型为 enum kvm_mr_change。__ kvm_set_memory_region 函数定义如下。

linux - 4.19.0/virt/kvm/kvm_main.c

```
int __ kvm_set_memory_region(struct kvm * kvm,
                const struct kvm_userspace_memory_region * mem) {
    struct kvm_memory_region old, new;
    struct kvm_memslots * slots = NULL;          // 新的 memslots
    enum kvm_mr_change change;
    new = old = * slot;                          // 获取原位上的旧 slot

    // 省略 new 的填充，根据 mem 进行填充即可
    // 省略对比 old/new 得出 kvm_mr_change 类型
```

```
    if (change == KVM_MR_CREATE) {
        new.userspace_addr = mem->userspace_addr;
        if (kvm_arch_create_memslot(kvm, &new, npages))
            goto out_free;
    }

    update_memslots(slots, &new);                  // 将 new 插入 slots
    install_new_memslots(kvm, as_id, slots);       //使用 RCU 将 kvm->memslots 设为 slots
}
```

该函数首先得到要插入位置的旧 memslot，与用户态传来的新 memslot 对比，得出 change 的类型。如果 QEMU 添加新的 memslot，那么就会进入 KVM_MR_CREATE 的分支，执行架构相关函数 kvm_arch_create_memslot 填充新的 memslot。准备好新的 memslot 后，就调用 update_memslots 函数将新 memslot 填入 slots 数组，并保持数组的排序(base_gfn 从大到小)，最终原子地将 slots 填入 kvm 中。KVM 维护 x86 **架构相关**的客户机物理页信息，包含 kvm_rmap_head 以及 kvm_lpage_info，代码如下。

linux-4.19.0/arch/x86/include/asm/kvm_host.h

```
struct kvm_arch_memory_slot {
    struct kvm_rmap_head * rmap[KVM_NR_PAGE_SIZES];
    struct kvm_lpage_info * lpage_info[KVM_NR_PAGE_SIZES - 1];
}
struct kvm_rmap_head {
    // 存储 spte 的地址，或存储 spte 链表 struct pte_list_desc 的地址
    unsigned long val;
};
struct kvm_lpage_info {
  int disallow_lpage;                      // 为 1 则不允许对应的 gfn 使用大页
};
struct pte_list_desc {
    u64 * sptes[PTE_LIST_EXT];             // PTE_LIST_EXT 为 3
    struct pte_list_desc * more;
};
```

针对 x86 架构，每个客户机的“虚拟”物理页都有相关的信息，保存在 memslot 的 arch 成员中。EPT 中最后一级页表可以是第 3 级页表，映射一个 1GB 的大页；可以是第 2 级页表，映射一个 2MB 的大页；也可以是通常情况的第 1 级页表，映射一个 4KB 的页。因此在 arch 结构体中，KVM 将 memslot 管理的这段“虚拟”物理内存按照不同大小的页面分割，共上述 3 种情况(1GB、2MB、4KB)，形成不同个数的页面(如 1GB 的内存区域按照 1GB 分割，则只有 1 页；按照 2MB 分割，则有 512 页)。下面代码填充每个页面的相关信息，分为 KVM_NR_PAGE_SIZES 种页面大小的情况。

linux-4.19.0/arch/x86/kvm/x86.c

```
int kvm_arch_create_memslot(memslot, npages) ->{
```

```
    // 遍历所有可能的页面大小,共 KVM_NR_PAGE_SIZES 种页面大小
    for (i = 0; i < KVM_NR_PAGE_SIZES; ++i) {          // 第 i + 1 级页表是最后一级页表
        lpages = gfn_to_index(slot->base_gfn + npages - 1,
                    slot->base_gfn, i + 1) + 1;     // 计算当前页面大小下的页面数
        slot->arch.rmap[i] =
            kvcalloc(lpages, sizeof( * slot->arch.rmap[i]), GFP_KERNEL);

        // 4KB 页不是大页,lpage_info 无须记录其信息
        // 故 lpage_info 数组长度为 rmap 长度减 1
        if (i == 0)
            continue;

        linfo = kvcalloc(lpages, sizeof( * linfo), GFP_KERNEL);
        slot->arch.lpage_info[i - 1] = linfo;
        // 省略是否可以使用大页的判断
        ... arch.lpage_info[i - 1][...].disallow_lpage = 1
    }
}
```

这里,每个“虚拟”物理页都有两个信息:①映射该 gfn 的所有 spte(EPT 页表项),保存在 arch.rmap 数组中,可以以 spte 链表的形式存在,每个页面都有对应的链表。该链表负责在某 HVA 处的页面被换出时,将所有与该 HVA 对应的 spte 置为无效。这种**反向映射**机制在 Linux 内核中也存在。②保存该 gfn 处是否可以使用大页,不做深入分析。

可以看到,KVM 已经保存了所有客户机“虚拟”物理内存页面的信息,存在于 memslots 成员中,KVM 维护 EPT 页表时将完全基于 memslots 数据结构。下面分析 KVM 如何维护 EPT 页表,与 MMU 硬件协同完成地址翻译。

2. 创建 vCPU 的虚拟 MMU

继续介绍 EPT 页表页的创建与管理,相关数据结构如图 3-16 所示。其中需要着重关注的是 struct kvm_mmu_page,它管理了一个 EPT 页表页的所有信息,vCPU 虚拟 MMU 的主要工作是维护 EPT 页表中每个页表页的 struct kvm_mmu_page。

为了管理 EPT 页表页,KVM 使用了 struct kvm_mmu_page 数据结构保存一个 EPT 页表页的相关信息,又使用 struct kvm_mmu 数据结构保存与内存管理相关的函数模拟硬件 MMU。每个 vCPU 都有一个虚拟的 MMU,且 MMU 的模拟依赖架构,因此保存在 struct kvm_vcpu_arch 数据结构中。注意区分,struct kvm_arch 保存了整个客户机的架构相关信息,而 struct kvm_vcpu_arch 保存了每个 vCPU 的架构相关信息。简而言之,虚拟 MMU 保存在 vCPU 的**架构相关**部分中,虚拟 MMU 的 root_hpa 指向 kvm_mmu_page(后文介绍)组成的页表。以下为这些数据结构。

linux-4.19.0/arch/x86/include/asm/kvm_host.h

```
struct kvm_vcpu_arch {
    unsigned long cr3;                  // 客户机 vCPU 的 cr3
    struct kvm_mmu mmu;                 // 进行 GPA -> HPA 翻译的虚拟 MMU
```

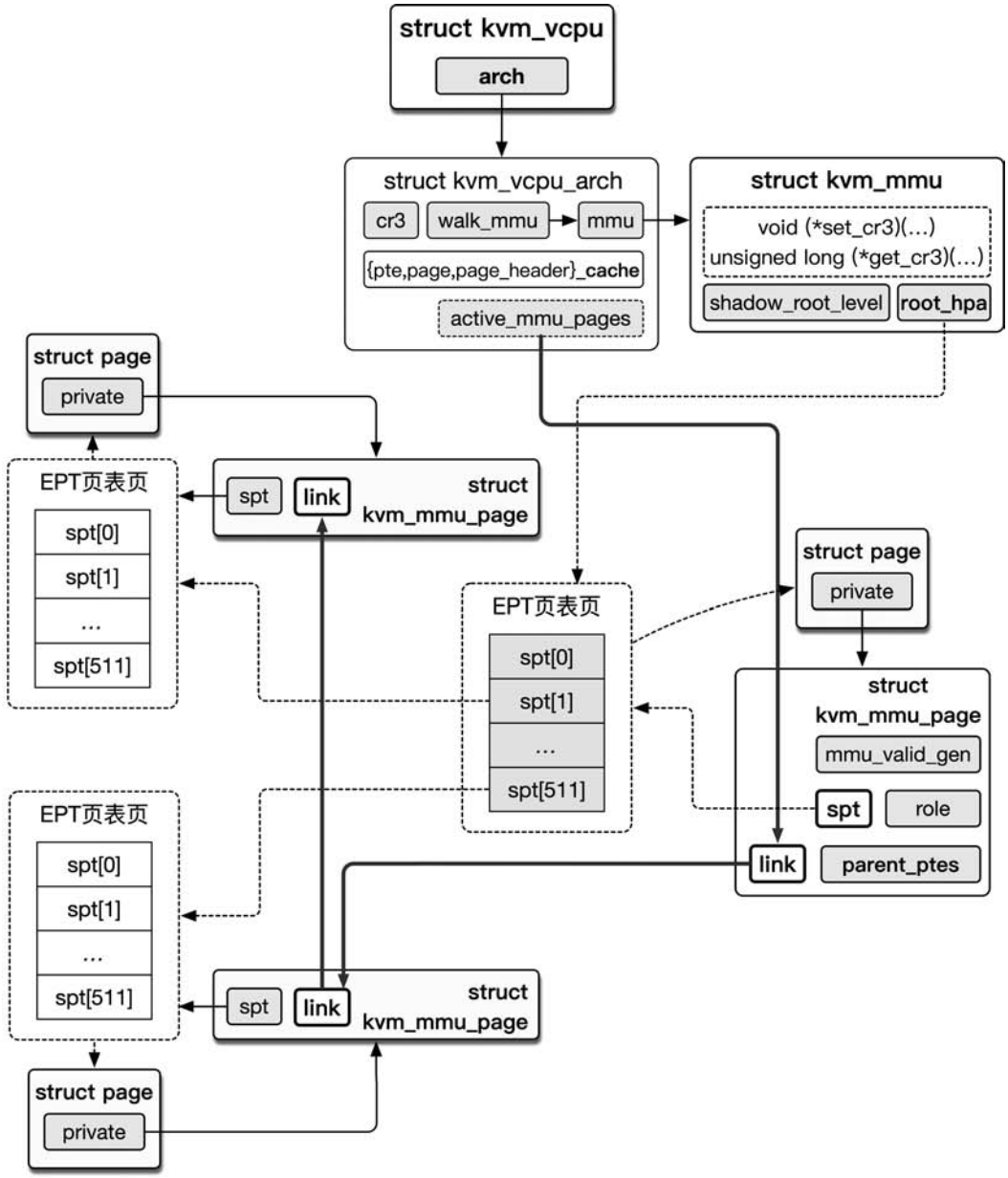


图 3-16　虚拟 MMU 相关数据结构

```
struct kvm_mmu *walk_mmu;                    // 当前正在工作的 MMU 的指针

struct kvm_mmu_memory_cache mmu_pte_list_desc_cache;    // pte 链表缓存池
struct kvm_mmu_memory_cache mmu_page_cache;             // 页表页缓存池
struct kvm_mmu_memory_cache mmu_page_header_cache;      // kvm_mmu_page 缓存池
```

```
    // EPT 页表页相关数据
    unsigned int n_used_mmu_pages, n_requested_mmu_pages, n_max_mmu_pages;

    // 维护此 struct kvm 的所有 EPT 页表页
    struct hlist_head mmu_page_hash[KVM_NUM_MMU_PAGES];          // EPT 页表页哈希表
    struct list_head active_mmu_pages;                           // 全部活跃的 EPT 页表页
    unsigned long mmu_valid_gen;                                 // 当前版本号
}

struct kvm_mmu_memory_cache {              // 通过提前分配页面形成缓存池,加快数据结构的分配
  int nobjs;                                                     // 缓存池中的缓存对象个数
  void * objects[KVM_NR_MEM_OBJS];                               // 缓存对象列表
};

struct kvm_mmu {
    void ( * set_cr3)(struct kvm_vcpu * vcpu, unsigned long root); //设置客户机 cr3
    unsigned long ( * get_cr3)(struct kvm_vcpu * vcpu);          // 读取客户机 cr3
    int ( * page_fault)(struct kvm_vcpu * vcpu, gva_t gva, u32 err,
      bool prefault);                                            // 缺页异常处理函数

    hpa_t root_hpa;                                              // EPT 基地址
    u8 root_level;                                               // GPT 级数
    u8 shadow_root_level;                                        // EPT 级数
    bool direct_map;                                             // 是否开启 EPT
}
```

在 KVM 中有一个全局变量 enable_ept 决定是否开启 EPT 模式。如果 enable_ept 为 1,则启用 EPT 模式,最终将全局变量 tdp_enabled(在 arch/x86/kvm/mmu.c 中)置为 true。事实上还需要读取 VMCS 的相关配置域才能决定是否将 enable_ept 置为 1,简洁起见本节省略这部分的介绍。下面是将 tdp_enable 置为 true 的代码。

linux-4.19.0/arch/x86/kvm/vmx.c

```
static bool __read_mostly enable_ept = 1;
static __init int hardware_setup(void) ->
    if (enable_ept) vmx_enable_tdp() ->
        kvm_enable_tdp() -> { tdp_enabled = true; }
```

在此之后创建 vCPU 时,将完成基于 EPT(即 tdp)的虚拟 MMU 初始化。虚拟 MMU 的初始化分为 kvm_mmu_create/kvm_mmu_setup 两步:创建和填充。调用链如下。

MMU 创建流程

```
kvm_vm_ioctl -> kvm_vm_ioctl_create_vcpu(kvm, id) -> {      //创建 vCPU 的 ioctl 调用
    vcpu = kvm_arch_vcpu_create(kvm, id) -> {
        vcpu = kvm_x86_ops -> vcpu_create(kvm, id) ->
            vmx_create_vcpu -> {
                kvm_vcpu_init -> kvm_arch_vcpu_init -> kvm_mmu_create(vcpu) {
                    vcpu -> arch.walk_mmu = &vcpu -> arch.mmu;
```

```
                    vcpu->arch.mmu.root_hpa = INVALID_PAGE;
                    alloc_mmu_pages() {
                        if (tdp_enabled) return;
                    }
                }
            }
        return vcpu;
    } // kvm_arch_vcpu_create

    kvm_arch_vcpu_setup(vcpu) -> kvm_mmu_setup(vcpu) -> {
        MMU_WARN_ON(VALID_PAGE(vcpu->arch.mmu.root_hpa)); // 当 vCPU 创建时调用
        kvm_init_mmu(vcpu, false) -> {
            ... // 省略除 tdp 类型 MMU 的初始化
            else if (tdp_enabled) {                          // 基于 tdp 的虚拟 MMU 的初始化
                init_kvm_tdp_mmu(vcpu) -> {
                    struct kvm_mmu * context = &vcpu->arch.mmu;

                    context->page_fault = tdp_page_fault;
                    context->set_cr3 = kvm_x86_ops->set_tdp_cr3;
                    context->direct_map = true;

                    // 判断 vCPU 的执行模式
                    if (!is_paging(vcpu)) {
                        context->...
                    }
                    else if (is_long_mode(vcpu)) {
                        context->...
                    }
                } // init_kvm_tdp_mmu
            } // if ...
        } // kvm_init_mmu
    } // kvm_mmu_setup

    kvm->vcpus[atomic_read(&kvm->online_vcpus)] = vcpu; // vCPU 创建完成
} // kvm_vm_ioctl_create_vcpu
```

虚拟 MMU 的初始化与 vCPU 的初始化绑定，即一个 vCPU 有一个虚拟 MMU。可以看到，虚拟 MMU 事实上包含了一组与 tdp 相关的页表管理函数，包括缺页异常处理函数 tdp_page_fault、页表基地址设置函数 set_tdp_cr3（即 vmx_set_cr3），以及虚拟 MMU 的相关属性的设置。完成虚拟 MMU 的创建后，下文介绍 KVM 如何维护 EPTP 和客户机 CR3。

EPT 页表页 kvm_mmu_page 与普通进程的页表页一样，在缺页异常中创建。这不是一般的缺页异常，而是客户机引发的 VM-Exit，是一个由 Intel 硬件提供的特性。该 VM-Exit 将使客户机暂停执行并进入 KVM，使得 KVM 有机会完善 EPT。为了配合硬件，KVM 需要将 EPT 的基地址写入 VMCS，让硬件 MMU 自动访问该 EPT 页表，当硬件 MMU 发现该页表存在不完整的情况时，将产生 VM-Exit，并调用虚拟 MMU 的相关函数。

接上文对 VM 文件描述符的 KVM_CREATE_VCPU 调用，QEMU 将对 vCPU 对应的文件描述符进行 KVM_RUN 调用，运行刚刚初始化并填充好的 vCPU，流程如下。

EPT 基地址设置流程

```
kvm_vcpu_ioctl -> kvm_arch_vcpu_ioctl_run -> vcpu_run ->
    for (;;) {
        if (kvm_vcpu_running(vcpu))
            r = vcpu_enter_guest(vcpu);
        else
            r = vcpu_block(kvm, vcpu);
        if (r <= 0)
            break;
    }

vcpu_enter_guest(vcpu) -> {
    r = kvm_mmu_reload(vcpu) -> {
        if (likely(vcpu->arch.mmu.root_hpa != INVALID_PAGE))      // EPT 已建立好
            return 0;
        return kvm_mmu_load(vcpu);
    }
    kvm_x86_ops->run(vcpu);                                       // 进入客户机
    r = kvm_x86_ops->handle_exit(vcpu);                           // 处理 VM-Exit
}

kvm_mmu_load(vcpu) -> {
    mmu_topup_memory_caches(vcpu);                    // 预分配 kvm_vcpu_arch 中的三个缓存

    mmu_alloc_roots -> mmu_alloc_direct_roots -> {                // 分配 EPT 第 4 级页表页
        struct kvm_mmu_page *sp;
        if (vcpu->arch.mmu.shadow_root_level >= PT64_ROOT_4LEVEL) { //四级页表
            sp = kvm_mmu_get_page(vcpu, 0, 0,
                vcpu->arch.mmu.shadow_root_level, 1, ACC_ALL);
            vcpu->arch.mmu.root_hpa = __pa(sp->spt);              // 设置 root_hpa
        }
    }

    kvm_mmu_load_cr3(vcpu) {
        if (VALID_PAGE(vcpu->arch.mmu.root_hpa)) {  // 如果 root_hpa 指向有效的 EPT
            vcpu->arch.mmu.set_cr3(vcpu->arch.mmu.root_hpa) { ->
                vmx_set_cr3(vcpu, cr3) -> {
                    eptp = construct_eptp(vcpu, cr3);
                    vmcs_write64(EPT_POINTER, eptp);              // 设置 EPT
                    guest_cr3 = kvm_read_cr3(vcpu);
                    vmcs_writel(GUEST_CR3, guest_cr3);            // 设置客户机 CR3
                } // vmx_set_cr3
            } // set_cr3
        } // if

        // 刷新 EPT 的 TLB
```

```
            kvm_x86_ops -> tlb_flush(vcpu, true) -> vmx_flush_tlb -> {
                ASM_VMX_INVEPT
                VMX_VPID_EXTENT_SINGLE_CONTEXT
                VMX_VPID_EXTENT_ALL_CONTEXT
            }
        } // kvm_mmu_load_cr3
    } // kvm_mmu_load
```

可以看到，vcpu_run 函数中出现了无限 for 循环，保证在运行 vCPU 退出后，完成 KVM 的处理继续运行 vCPU。在该循环中，运行 vCPU 前调用 kvm_mmu_reload 函数，如果 root_hpa 尚未初始化(指向 INVALID_PAGE)，则调用 kvm_mmu_load 函数初始化 root_hpa，其初始化工作主要包括：①调用 mmu_topup_memory_caches 函数保证 arch 中的 3 个 cache 充足；②为 root_hpa 分配一个 kvm_mmu_page，作为 EPT 的第 4 级页表页，页表页的分配将在下文介绍；③根据上一步分配的 EPT 第 4 级页表页的物理地址，创建 EPTP，写入 VMCS 的 EPTP 域中，EPTP 域的详细文档见 Intel SDM 的卷 3 第 24.6.11 节；还要从 arch 结构中读取客户机 CR3，写入 VMCS 的 GUEST_CR3 域中。这样在物理 CPU 进入非根模式后，就可以使用该 EPTP 进行 GPA 到 HPA 的翻译；④刷新 TLB，有三种方式，不详细介绍。综上，KVM 已经准备好进入非根模式，执行 kvm_x86_ops-> run 函数。下面介绍 EPT 页表页及其创建过程，EPT 页表的创建和维护过程均在 kvm_x86_ops-> handle_exit 函数中进行。

3. EPT 的建立

KVM 从 QEMU 得到了需要 GPA 到 HPA 翻译的所有 kvm_memory_slots，设置好 EPTP 后，MMU 硬件就可以截获 GPA 地址的访问，使客户机退出到 KVM 中，建立 GPA 相关的 EPT 页表。这里介绍 EPT 的建立与管理，首先介绍管理 EPT 页表页的数据结构 kvm_mmu_page，它定义如下。

linux - 4.19.0/arch/x86/include/asm/kvm_host.h

```
struct kvm_mmu_page {                          // 页表页描述结构
    struct list_head link;                     // active_mmu_pages 链表中的节点
    struct hlist_node hash_link;               // 哈希表中的节点
    union kvm_mmu_page_role role;              // 该 EPT 页表页的属性
    u64 * spt;                                 // 页表页的宿主机虚拟地址
    struct kvm_rmap_head parent_ptes;          // 指向此页表页的 pte 的链表
    unsigned long mmu_valid_gen;               // 此页表页版本号
}
```

该结构维护了一个 spt 指针关联的信息，spt 是指向页表页的指针，是一个 HVA。spt 指向的 4KB 大小的页面即是 EPT 页表页，保存了 512 个页表项。KVM 在 struct kvm_arch 中保存了一个 active_mmu_pages 链表，将所有的 kvm_mmu_page 链接起来，可以当页表页版本号 mmu_valid_gen 与 arch 中的 mmu_valid_gen 不同时，释放页表页(通过其操

作接口 kvm_mmu_free_page 函数)。这样 KVM 就做到了内存占用尽可能小,这和 **mmu.txt** 文档中 KVM 内存虚拟化设计原则相符合。

创建页表页的时机在于发生 EPT Violation 时,CPU 进入根模式运行 KVM。在这里,KVM 执行如下虚拟 MMU 的缺页异常处理函数。

linux-4.19.0/arch/x86/kvm/mmu.c

```
vcpu_enter_guest -> kvm_x86_ops -> handle_exit(vcpu) ->
    vmx_handle_exit -> kvm_vmx_exit_handlers[exit_reason](vcpu);

static int ( * const kvm_vmx_exit_handlers[])(struct kvm_vcpu * vcpu) = {
    [EXIT_REASON_EPT_VIOLATION]         = handle_ept_violation,
}
enum {                                  // 各类返回值
    RET_PF_RETRY = 0,                   // 返回客户机重新执行引起 EPT Violation 的指令
    RET_PF_EMULATE = 1,                 // 执行模拟
    RET_PF_INVALID = 2,                 // 无效的模拟,继续执行真实的缺页异常处理过程
};
static int handle_ept_violation(struct kvm_vcpu * vcpu) {
    vcpu->arch.exit_qualification = vmcs_readl(EXIT_QUALIFICATION);
    error_code = PFERR_ * _MASK;        // 从 exit_qualification 获得
    gpa = vmcs_read64(GUEST_PHYSICAL_ADDRESS);

    return kvm_mmu_page_fault(vcpu, gpa, error_code, NULL, 0) {
        r = RET_PF_INVALID;
        if (unlikely(error_code & PFERR_RSVD_MASK)) {
            r = handle_mmio_page_fault(vcpu, cr2, direct);   // MMIO 模拟
            if (r == RET_PF_EMULATE)
                goto emulate;           // 执行模拟
        }
        if (r == RET_PF_INVALID)
            vcpu->arch.mmu.page_fault(vcpu, gpa, error_code, false) ->
                tdp_page_fault(vcpu, gpa, error_code, prefault)
        if (r == RET_PF_RETRY)
            return 1;                   // 重新执行引起 EPT Violation 的指令
    }
}
```

退出原因是 EXIT_REASON_EPT_VIOLATION,于是调用 handle_ept_violation 函数。VMCS 保存了该缺页异常的相关信息,包括造成缺页异常的 gpa、缺页异常的类型 error_code 等。如果 error_code 表示需要处理 MMIO 类型的 EPT Violation,则调用 handle_mmio_page_fault 函数。在 EPT 页表缺页的情况下,调用 tdp_page_fault 函数完善 EPT。这里忽略所有与大页相关的代码,感兴趣的读者可以在介绍的基础上研究大页相关代码,进行“增量式”学习。tdp_page_fault 函数主要代码如下。

linux-4.19.0/arch/x86/kvm/mmu.c

```
static int tdp_page_fault(struct kvm_vcpu * vcpu, gva_t gpa, u32 error_code,
```

```
                bool prefault)
{
    kvm_pfn_t pfn;                                          // GPA 对应的 HPA
    int level;              // 最后一级页表的 level,1GB 的大页是 3,2MB 的大页是 2,4KB 页是 1
    gfn_t gfn = gpa >> PAGE_SHIFT;
    int write = error_code & PFERR_WRITE_MASK;

    mmu_topup_memory_caches(vcpu) -> {                      // 预分配缓存池
        mmu_topup_memory_cache(&vcpu->arch.mmu_pte_list_desc_cache,
                pte_list_desc_cache, 8 + PTE_PREFETCH_NUM);
        mmu_topup_memory_cache_page(&vcpu->arch.mmu_page_cache, 8);
        mmu_topup_memory_cache(&vcpu->arch.mmu_page_header_cache,
                mmu_page_header_cache, 4);
    }

    level = mapping_level(vcpu, gfn, &force_pt_level);      // level = 1
    if (fast_page_fault(vcpu, gpa, level, error_code))      // 用于脏页跟踪
        return RET_PF_RETRY;

    if (try_async_pf(vcpu, prefault, gfn, gpa, &pfn, write, &map_writable))
        return RET_PF_RETRY;

    spin_lock(&vcpu->kvm->mmu_lock);                        // 获取 EPT 锁
    if (mmu_notifier_retry(vcpu->kvm, mmu_seq))
        goto out_unlock;
    if (make_mmu_pages_available(vcpu) < 0)
        goto out_unlock;
    r = __direct_map(vcpu, write, map_writable, level, gfn, pfn, prefault);
    spin_unlock(&vcpu->kvm->mmu_lock);                      // 释放 EPT 锁

    return r;
out_unlock:
    spin_unlock(&vcpu->kvm->mmu_lock);
    kvm_release_pfn_clean(pfn);
    return RET_PF_RETRY;
}
```

tdp_page_fault 函数较为复杂，涉及多个 Linux 内核子系统，下面分步介绍。

(1) 该函数首先填充三个缓存池，每次预分配都分别增加 16、8、4 个预备的对象。这三个缓存池分别用于 pte 链表(反向映射中一个 gfn 对应的所有 pte 的链表，以及一个 pte 的 parent_ptes 链表)、EPT 页表页、EPT 页表页头(即描述 EPT 页表页的 struct kvm_mmu_page)的分配，这三类对象的分配在 KVM 中很频繁，因此缓存池能够通过合并分配提高分配效率。

(2) 这里忽略大页管理系统，假定 EPT 中不映射大页，mapping_level 函数将返回 1。接下来，尝试缺页异常处理的快速路径，调用 fast_page_fault 函数。此函数和 KVM 脏页管理系统有关，回忆当 QEMU 注册 memslot 时，可以使用 KVM_MEM_LOG_DIRTY_

PAGES标志，它表示需要对这段内存进行脏页跟踪，多用于QEMU虚拟机热迁移的实现。当QEMU使用该标志进行内存注册时，KVM将会把这段内存对应的所有EPT表项置为只读，这涉及内存页的反向映射系统。于是，该页表项指向的物理页已经分配，只需要将该页表项修改为可写就可继续正常执行客户机代码，而不再产生EPT Violation，不需要执行后面的真实缺页异常处理。

(3) try_async_pf函数负责检查客户机访问的GPA是否在KVM的memslots中，如果在，则分配宿主机物理页面，得到pfn。EPT将GPA翻译为HPA，那么为了填充EPT表项，一个客户机物理页(GPA)必将对应一个宿主机物理页(HPA)。但是，如果该宿主机内存页已经换出到磁盘上，则会使客户机vCPU停滞较长的一段时间。为此，KVM实现了异步缺页异常，当一个vCPU访问了被换出的宿主机物理页(这里借助了宿主机Linux内核的内存管理相关接口，不做深入介绍)时，则会将vCPU挂起，并执行另一个vCPU，等待被换出的页加载到内存中。加载完毕后，被挂起的vCPU则会从try_async_pf函数中返回，重新执行引起缺页异常的客户机指令。该函数的另一个任务是实现MMIO，检查参数gfn是否在KVM的memslots中，如果不在，则向pfn中写入KVM_PFN_NOSLOT，最后进入__direct_map函数的set_mmio_spte函数中，将EPT中这段客户机物理内存对应的部分标为特殊值，以后的读写都会引起EPT Misconfiguration的VM-Exit。至此，KVM获得了pfn(类型为kvm_pfn_t，表示HPA)，交由下一步构建EPT表项、填充EPT使用。

(4) 进入修改EPT的代码区，由于多个vCPU线程以及MMU通知器(MMU Notifier，当Linux内核将虚拟内存页换出到磁盘上时将收到通知，KVM也注册了一个这样的通知器，其具体作用是在Linux内核换出客户机内存页时修改EPT)可能会同时修改EPT，需要加kvm的mmu_lock锁。mmu_notifier_retry函数让MMU通知器线程优先执行，放弃vCPU的EPT填充。接下来，make_mmu_pages_available函数将无用的EPT页表页释放，与之前填充EPT页的分配缓存池相互照应。这里也用到了反向映射系统。

__direct_map函数实际填充了EPT四级页表，其定义如下。

linux-4.19.0/arch/x86/kvm/mmu.c

```
static int __direct_map(struct kvm_vcpu *vcpu, int write, int map_writable,
            int level, gfn_t gfn, kvm_pfn_t pfn, bool prefault) {
    struct kvm_shadow_walk_iterator iterator;
    struct kvm_mmu_page *sp;
    int emulate = 0;
    gfn_t pseudo_gfn;

    for_each_shadow_entry(vcpu, (u64)gfn << PAGE_SHIFT, iterator) {
        if (iterator.level == level) {
            emulate = mmu_set_spte(vcpu, iterator.sptep, ACC_ALL,
                        write, level, gfn, pfn, prefault,
                        map_writable);
            direct_pte_prefetch(vcpu, iterator.sptep);
            break;
        }
```

```
        if (!is_shadow_present_pte( * iterator.sptep)) {
            u64 base_addr = iterator.addr;
            base_addr &= PT64_LVL_ADDR_MASK(iterator.level);
            pseudo_gfn = base_addr >> PAGE_SHIFT;
            sp = kvm_mmu_get_page(vcpu, pseudo_gfn, iterator.addr,
                        iterator.level - 1, 1, ACC_ALL);
            link_shadow_page(vcpu, iterator.sptep, sp);
        }
    }
    return emulate;      // 可能返回 RET_PF_EMULATE
}
```

该函数根据前面步骤得到的 gfn(客户机物理页号)以及 pfn(宿主机物理页号)完成 EPT 中 gfn 对应部分的建立。for_each_shadow_entry 宏负责遍历四级页表,其参数是客户机物理地址(gfn<<PAGE_SHIFT)和遍历光标 iterator。熟悉 C++11 语法的读者知道,iterator 承载了指针的作用,且方便了对一个数据结构的遍历。而此处的 iterator(即读取 EPT 页表时指向 EPT 的指针)中存储了读取页表时的相关参数,方便了 EPT 的操作。详细内容见注释,此处仅关注其中的 level、addr 和 sptep。

linux-4.19.0/arch/x86/kvm/mmu.c

```
struct kvm_shadow_walk_iterator {
    u64 addr;              // 当前查询 EPT 的 GPA,在整个过程中不变,即 gfn << PAGE_SHIFT
    hpa_t shadow_addr;     // 当前 for 循环遍历到的页表页的物理地址
    u64 * sptep;           // 当前 for 循环遍历到的页表项
    int level;             // 当前 for 循环遍历到的页表页的级数,即 4、3、2、1
    unsigned index;        // 当前 for 循环遍历到的页表项在本级页表中的索引
};
```

在此处的讨论中,for_each_shadow_entry 宏会执行 4 次,由于忽略了大页,EPT 共有四级页表。分两种情况:①遍历到的页表页是页表的中间节点,如果当前页表项 * iterator.sptep 不存在,则调用 kvm_mmu_get_page 函数获取一个 kvm_mmu_page,并调用 link_shadow_page 函数将当前 sptep 指向新的页表页;②如果遍历到的是最后一级页表,则调用 mmu_set_spte 函数将最后一级页表项指向新的 pfn。这样就建立了映射 gfn 到 pfn 的 EPT。

(5) 最终释放 mmu_lock,如果更改 EPT 页表过程中的某一步失败,还需要释放由 try_async_pf 函数分配的 pfn 处的宿主机物理页。至此,KVM 完成了 gpa 对应的 EPT 四级页表的填充。

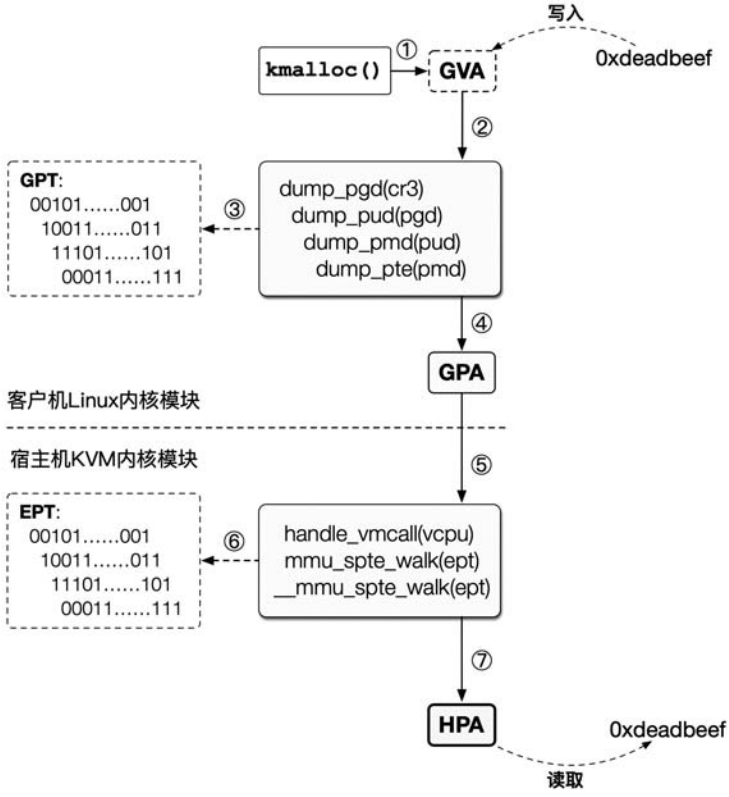
3.3.4 实验：将 GVA 翻译为 HPA

1. 实验概述

作为内存虚拟化的总结以及 KVM 内存虚拟化源码章节的拓展,本节进行 GVA 到 HPA 的翻译实验。内存虚拟化的核心是地址翻译,即将某一个地址空间的地址转换为下

层地址空间的地址。如前文所述，地址翻译由 MMU 硬件完成，首先使用客户机进程页表 GPT 将 GVA 翻译为 GPA，再由扩展页表 EPT 将 GPA 翻译为 HPA。由于无法观察硬件的地址翻译过程，于是本节借助内核提供的页表访问接口，通过编写软件模拟 MMU 的功能。

为了证明内存翻译代码运行的正确性，首先在 GVA 处写入一个 int 类型的变量，并在最后得到的 HPA 处对该 int 类型变量进行读取，如果写入的变量和读到的变量值相同，那么证明地址翻译正确。本实验实现的软件 MMU 分为两部分：①客户机中的地址翻译模块作为客户机中运行的内核模块，首先在 GVA 处写入一个 int 变量 **0xdeadbeef**，再通过读取 GPT 的方式将 GVA 翻译成 GPA，最后通过超级调用将 GPA 传递给宿主机操作系统；②宿主机中的 KVM 内核模块截获到该超级调用，得到客户机传来的 GPA，通过读取 EPT 的方式进一步翻译成 HPA。为了读取 HPA 处的变量，还需要使用内核提供的接口做一次 HPA 到 HVA 的转化，这是因为分页模式开启，访存指令中的地址均是 HVA，无法使用 HPA 直接访问物理内存。最终读取 HVA 处的变量，将读取到的值与客户机写入的值进行比较，如果也是 **0xdeadbeef**，则能够证明地址翻译的正确性。下文分别介绍客户机的地址翻译与宿主机的地址翻译，形成一个整体，使读者了解地址翻译的流程，流程如图 3-17 所示。



注：①调用 kmalloc 函数得到 **GVA**；②查询 GPT；③打印 GPT 相关表项；④查询 GPT，得到 **GPA** ；⑤通过超级调用将 GPA 传入 KVM，查询 EPT；⑥打印 EPT 相关表项；⑦查询 EPT，得到 **HPA**。

图 3-17　GVA 到 HPA 翻译实验流程

首先说明实验的运行环境：宿主机 CPU 型号是 Intel Core i7-6500U，频率 2.50GHz，QEMU 为客户机提供了与宿主机相同型号的 CPU。宿主机物理内存大小为 7.5GB，客户机物理内存为 4GB。本节宿主机和客户机均使用 Linux v4.19.0 内核，并使用从源码编译的 QEMU v4.1.1。再说明实验准备，读者应该事先制作好一个客户机磁盘镜像，作为 QEMU 的-hda 参数进行读取，使用 QEMU 启动一个客户机。实验相关过程与脚本见代码仓库[①]。

2. 客户机中的地址转换：GVA 到 GPA

为了读取 Linux 操作系统的页表，需要在内核态编写代码。使用 Linux **内核模块**是编写内核代码的一种简单的方式。内核模块的源码可以仅由一个.c 文件组成，在实验里是 gpt-dump.c，只需要编写一个 Makefile(编译命令文件)对 gpt-dump.c 进行编译，得到 gpt-dump.ko 文件，再使用命令 sudo insmod gpt-dump.ko 即可将内核模块插入内核。内核模块插入内核后，就会调用 gpt-dump.c 中定义的 init_module 函数；而使用命令 sudo insmod gpt-dump.ko 将内核模块移出内核时，则会调用 cleanup_module 函数。在 init_module 函数中即可编写内核代码在内核态运行，可以访问页表。

虚拟地址和物理地址在 Linux 内核中均使用 64 位的变量表示，而在 Intel Core i7 系统中，虚拟地址空间为 48 位，共 256TB 大小，物理地址空间为 52 位，共 4PB(4096TB)大小。一个页表项的大小为 64 位，即 8 字节，一个页表页有 512 个页表项，页表页大小为 4KB，故需要虚拟地址中的 $\log_2(512)=9$ 位索引每级的页表页。Linux 内核使用 4 级页表，用虚拟地址的第 47:39 位索引第 4 级页表，第 38:30 位索引第 3 级页表，第 29:21 位索引第 2 级页表，第 20:12 位索引第 1 级页表。这样就形成了前文所述的 9+9+9+9 形式的四级页表。

虚拟地址使用第 47:12 位存储页表的索引，第 11:0 位存储虚拟地址在页中的偏移，因此查询页表只使用了虚拟地址的前 48 位，即访问虚拟地址空间仅使用了 48 位的虚拟地址。此处定义宏 UL_TO_VADDR 以及 VADDR_PATTERN 打印虚拟地址，包含页表的索引位(共 36 位)，以及偏移(共 12 位)。部分相关宏定义如下。

gpt-dump/gpt-dump.c

```
#define TBYTE_TO_BINARY(tbyte)  \
  ((tbyte) & 0x04 ? '1' : '0'),    \
  ((tbyte) & 0x02 ? '1' : '0'), ((tbyte) & 0x01 ? '1' : '0')
#define UL_TO_PTE_OFFSET(ulong) \
  TBYTE_TO_BINARY((ulong) >> 9), TBYTE_TO_BINARY((ulong) >> 6), \
  TBYTE_TO_BINARY((ulong) >> 3), TBYTE_TO_BINARY((ulong))
#define UL_TO_PTE_INDEX(ulong) \
  TBYTE_TO_BINARY((ulong) >> 6), TBYTE_TO_BINARY((ulong) >> 3), TBYTE_TO_BINARY((ulong))
#define UL_TO_VADDR(ulong) \
  UL_TO_PTE_INDEX((ulong) >> 39), UL_TO_PTE_INDEX((ulong) >> 30), \
  UL_TO_PTE_INDEX((ulong) >> 21), UL_TO_PTE_INDEX((ulong) >> 12), \
```

① 代码仓库地址：https://github.com/GiantVM/book，包含了本书的实验代码。

```
  UL_TO_PTE_OFFSET((ulong))

#define TBYTE_TO_BINARY_PATTERN    "%c%c%c"
#define PTE_INDEX_PATTERN \
  TBYTE_TO_BINARY_PATTERN TBYTE_TO_BINARY_PATTERN TBYTE_TO_BINARY_PATTERN " "
#define VADDR_OFFSET_PATTERN \
  TBYTE_TO_BINARY_PATTERN TBYTE_TO_BINARY_PATTERN \
  TBYTE_TO_BINARY_PATTERN TBYTE_TO_BINARY_PATTERN
#define VADDR_PATTERN \
  PTE_INDEX_PATTERN PTE_INDEX_PATTERN \
  PTE_INDEX_PATTERN PTE_INDEX_PATTERN \
  VADDR_OFFSET_PATTERN

int init_module(void) ->
    print_ptr_vaddr(ptr);
```

由于内核尚没有将变量转化为二进制表示的函数，此处编写 pr_info 函数，接收不同的"%c"格式化字符串和'0'/'1'字符的组合来打印虚拟地址、物理地址以及页表项。以虚拟地址的打印为例，首先定义输出 3 个位的字符组合，即 TBYTE_TO_BINARY，还有对应的打印三个 char 类型的格式化字符串 TBYTE_TO_BINARY_PATTERN。接下来，需要将虚拟地址的低 12 位（即偏移的 12 个位）全部打印出来。继续对代表虚拟地址的 ulong（64 位的变量）使用 TBYTE_TO_BINARY，得到其低 3 位；再将 ulong 右移 3 位，并使用宏 TBYTE_TO_BINARY，得到其 5:3 位，以此类推，可以得到所有形式的'0'/'1'字符组合，包括宏 UL_TO_PTE_OFFSET 负责输出 12 位，UL_TO_PTE_INDEX 负责输出 9 位。对于 pr_info 的格式化字符串，实现方式类似，只需在合适的位置加上空格，便于观察。在内核模块初始化函数中，首先调用 print_ptr_vaddr 打印虚拟地址，输出如下。

gpt - dump/gpt - dump.txt

```
Value at GVA: 0xdeadbeef
 GPT PGD index: 304
 GPT PUD index: 502
 GPT PMD index: 469
 GPT PTE index: 163
         304       502       469       163
 GVA [PGD IDX] [PUD IDX] [PMD IDX] [PTE IDX] [   Offset   ]
 GVA 100110000 111110110 111010101 010100011 011001011000
```

可以看到四级页表的四个索引值，以及页内偏移的二进制表示。为了获得一个 GVA，在模块初始化函数中，首先调用 kmalloc 函数生成一个 int 类型变量的指针，由于内核模块运行在客户机内核中，所以该指针包含一个 GVA。在上面的代码片段中，客户机内核模块在该指针处写入数字：**0xdeadbeef**，期望在 GVA 对应的 HVA 处读到该数字。在输出中可以看到四级页表的索引，得知在页表页中应该读取第几个页表项。接下来，内核模块找到客户机内核线程的 CR3，并将其传入页表打印函数。下面是客户机内核模块的相关代码。

gpt - dump/gpt - dump.c

```
/* static vals */
unsigned long vaddr, paddr, pgd_idx, pud_idx, pmd_idx, pte_idx;

int init_module(void) ->
    dump_pgd(current->mm->pgd, 1);

void dump_pgd(pgd_t *pgtable, int level) {
    unsigned long i;
    pgd_t pgd;
    for (i = 0; i < PTRS_PER_PGD; i++) {
        pgd = pgtable[i];

        if (pgd_val(pgd) && (i == pgd_idx)) {
            if (pgd_large(pgd))
                { pr_info("Large pgd detected! return"); break; }

            if (pgd_present(pgd)) {
                // 调用__pa 检查页表页起始地址
                pr_pte(__pa(pgtable), pgd_val(pgd), i, level);
                dump_pud((pud_t *) pgd_page_vaddr(pgd), level + 1);
            }
        }
    }
}
void dump_pud(pud_t *pgtable, int level);
void dump_pmd(pmd_t *pgtable, int level);
void dump_pte(pte_t *pgtable, int level);
```

current-> mm-> pgd 是当前进程 current 的 CR3，指向 PGD(Page Global Directory，页全局目录)页表页的起始地址。dump_pgd 函数负责遍历 pgd 页表页的 PTRS_PER_PGD 个页表项，这里是 512 个页表项。pgd 是一个 pgd_t 类型的变量，内核还提供了类似的 pud_t、pmd_t、pte_t 数据结构表示每级页表的页表项，以及操作这些数据结构的接口。此处使用这些接口获取页表项的含义，如 pgd_val 函数返回该页表项的值，即该页表项对应的 unsigned long 变量；pgd_present 函数检查该页表项的第 0 位，返回该页表项是否有效；pgd_large 函数返回该页表项是否指向 1GB 的大页。本节忽略大页的情况，读者可以使用内核参数关闭大页。这里定义了全局变量保存的虚拟地址和对应的物理地址，以及各级页表索引。pgd_idx 表示从 64 位虚拟地址中获得的 PGD 页表索引。接下来，如果 PGD 页表页的第 pgd_idx 个页表项存在，那么调用 pr_pte 函数打印该页表项，此函数定义如下。

gpt - dump/gpt - dump.c

```
const char *PREFIXES[] = {"PGD", "PUD", "PMD", "PTE"};
static inline void pr_pte(unsigned long address, unsigned long pte,
                          unsigned long i, int level) {
    if (level == 1)
```

```
        pr_cont(" NEXT_LVL_GPA(CR3)   =   ");
    else
        pr_cont(" NEXT_LVL_GPA( % s)   =   ", PREFIXES[level - 2]);
    pr_cont(PTE_PHYADDR_PATTREN, UL_TO_PTE_PHYADDR(address >> PAGE_SHIFT));
    pr_cont(" +   64 *  % - 3lu\n", i);
    pr_sep();     // 打印换行符
    pr_cont(" % - 3lu: % s " PTE_PATTERN"\n",
            i, PREFIXES[level - 1], UL_TO_PTE(pte));
}
```

参数 address 表示本级页表页的起始地址，从上一级页表项获得，pte 表示本级页表项。宏 PTE_PATTREN 用于打印一个页表项。继续查询下一级页表的函数调用链为 dump_pgd→dump_pud→dump_pmd→dump_pte，其中每一步的逻辑大致相同。address 和 pte 的打印结果如下。

gpt - dump/gpt - dump.txt

```
NEXT_LVL_GPA(CR3)   =   00000000000000000000001100001100000001100   +   64 * 304

304: PGD 000000000000 0000000000000000000000011111110110010011 000001100111
NEXT_LVL_GPA(PGD)   =   00000000000000000000000011111110110010011   +   64 * 502

502: PUD 000000000000 0000000000000000000000011111110110010100 00000110011
NEXT_LVL_GPA(PUD)   =   00000000000000000000000011111110110010100   +   64 * 469

469: PMD 000000000000 0000000000000000000000011101010101100100 000001100011
NEXT_LVL_GPA(PMD)   =   00000000000000000000000011101010101100100   +   64 * 163

163: PTE 100000000000 0000000000000000000000011101010101010100011 000001100011
GPA             =       00000000000000000000000011101010101010100011 011001011000
```

从加粗的部分可以看到，客户机内核模块对页表页的地址调用__pa 函数获得其物理地址，和上一级页表项中保存的下一级页表页的物理地址完全相同，这符合预期。最终，就可以从 PTE 中获得物理地址，为了验证正确性，客户机内核模块对 vaddr 调用__pa 函数，打印出 GPA 开头的行，具体在 print_pa_check 函数中实现，代码如下。

gpt - dump/gpt - dump.c

```
static inline void print_pa_check(unsigned long vaddr) {
    paddr = __pa(vaddr);
    pr_info(" GPA             =       " PADDR_PATTERN "\n", UL_TO_PADDR(paddr));
}
```

可以看到，和之前获得的 PTE 中的物理地址相同，说明读取页表的结果正确。其中低 12 位是页内偏移，物理地址和虚拟地址中的页内偏移完全相同。最后，客户机内核模块调用 kvm_hypercall1(22, paddr)函数，将 paddr 传给 KVM。

3. KVM 中的地址转换：GPA 到 HPA

KVM 负责维护 EPT，具有读取 EPT 的权限。本实验修改宿主机内核的 KVM 模块，增加一个超级调用处理函数，接收从客户机传来的 GPA，模拟 GPA 到 HPA 的翻译。在 CPU 虚拟化章节中提到，当客户机执行了一个敏感非特权指令时，会引起 CPU 的 VM-Exit，CPU 的执行模式从非根模式转换为根模式，并进入 KVM 的 VM-Exit 处理函数。kvm_hypercall1 函数最终执行 vmcall 指令，陷入 KVM，KVM 得知 VM-Exit 的原因是客户机执行了 vmcall 指令，编号为 EXIT_REASON_VMCALL，于是调用如下 handle_vmcall 处理函数。

linux-4.19.0/arch/x86/kvm/vmx.c

```
static int ( * const kvm_vmx_exit_handlers[])(struct kvm_vcpu * vcpu) =
{
    [EXIT_REASON_VMCALL] = handle_vmcall,
}
static int handle_vmcall(struct kvm_vcpu * vcpu)
{
    return kvm_emulate_hypercall(vcpu);
}
```

函数 handle_vmcall 调用超级调用模拟函数 kvm_emulate_hypercall，代码如下。

linux-4.19.0/arch/x86/kvm/x86.c

```
int kvm_emulate_hypercall(struct kvm_vcpu * vcpu)
{
    unsigned long nr, a0, a1, a2, a3, ret;
    nr = kvm_register_read(vcpu, VCPU_REGS_RAX);
    a0 = kvm_register_read(vcpu, VCPU_REGS_RBX);
    ..
    switch (nr) {
    case KVM_HC_DUMP_SPT:
        print_gpa_from_guest(a0);
        mmu_spte_walk(vcpu, pr_spte);
        ret = 0;
        break;
    ..
    }
}
```

其中，nr 是 kvm_hypercall1 函数的第一个参数，a0 是第二个参数。nr 表示应该调用哪个超级调用模拟函数，定义 KVM_HC_DUMP_SPT 为 22，表示打印客户机内核线程页表对应的 EPT。首先调用 print_gpa_from_guest 函数打印客户机传来的 GPA，并从 GPA 中获取每级 EPT 页表的索引，保存在全局变量 pxx_idx[4]中，print_gpa_from_guest 函数的打印结果如下。

gpt - dump/ept - dump.txt

```
EPT PGD index: 0
EPT PUD index: 0
EPT PMD index: 469
EPT PTE index: 163
      0       0       469       163
GPA [PGD IDX] [PUD IDX] [PMD IDX] [PTE IDX] [   Offset   ]
GPA 000000000 000000000 111010101 010100011 011001011000
```

可以看到,此处的 GPA 与在客户机中读取 GPT 得到的 GPA 相同,说明客户机内核模块的超级调用成功。接下来调用 mmu_spte_walk 函数遍历此 vCPU 的 EPT,在代码中称作 spt,这是为了和影子页表共用一套代码。传入 mmu_spte_walk 函数的参数有 vcpu,以及遍历到一个页表项时所调用函数的指针 pr_spte,负责打印页表项。遍历代码如下。

linux - 4.19.0/arch/x86/kvm/mmu.c

```
unsigned long gpa_from_guest, pxx_idx[PT64_ROOT_4LEVEL];
void mmu_spte_walk(struct kvm_vcpu * vcpu, inspect_spte_fn fn)
{
    int i;
    struct kvm_mmu_page * sp;
    if (!VALID_PAGE(vcpu->arch.mmu.root_hpa))
        return;
    if (vcpu->arch.mmu.root_level >= PT64_ROOT_4LEVEL) {
        hpa_t root = vcpu->arch.mmu.root_hpa;
        sp = page_header(root);
        __mmu_spte_walk(vcpu, sp, fn, 1);
    }
}
static void __mmu_spte_walk(struct kvm_vcpu * vcpu, struct kvm_mmu_page * sp,
                inspect_spte_fn fn, int level)
{
    int i;
    for (i = 0; i < PT64_ENT_PER_PAGE; ++i) {
        u64 * ent = sp->spt;
        if (i == pxx_idx[PT64_ROOT_4LEVEL - (5 - level)] &&
            is_shadow_present_pte(ent[i]))
        {
            struct kvm_mmu_page * child;
            fn(__pa(ent), ent[i], i, level);            // 调用__pa 检查页表页起始地址

            if (!is_last_spte(ent[i], 5 - level)) {     // 继续下一级遍历
                child = page_header(ent[i] & PT64_BASE_ADDR_MASK);
                __mmu_spte_walk(vcpu, child, fn, level + 1);
            } else {
                if (is_large_pte(ent[i]))               // 遇到大页
                    print_huge_pte(ent[i]);
                else                                    // 未遇到大页
```

```
                    print_pte(ent[i]);
                }
            }
        }
    }
```

如 3.3.3 节所述，vcpu-> arch. mmu. root_hpa 保存了 EPT 的基地址，初始化时被设为 INVALID_PAGE。mmu_spte_walk 函数首先判断 vcpu-> arch. mmu. root_hpa 是否是无效页 INVALID_PAGE，如果是，则说明 vCPU 对应的 EPT 尚未建立，无法遍历。如果 EPT 的级数大于 PT64_ROOT_4LEVEL，则调用递归函数__ mmu_spte_walk 遍历页表。

page_header 函数返回一个 hpa_t 变量所指向页表页的 kvm_mmu_page 结构的指针。于是，KVM 将 EPT 第 4 级页表页的 kvm_mmu_page 结构传入__ mmu_spte_walk 函数，并且从 level=1 开始遍历 EPT。

和查询 GPT 一样，KVM 遍历页表页中的每一个页表项，如果页表项的索引等于之前 print_gpa_from_guest 函数中获得的 pxx_idx 中对应的索引，那么此页表项就是目标页表项，并将它传入 fn 函数进行打印。如果查询到的页表项不是最后一级，则继续递归调用__ mmu_spte_walk 函数查询下一级页表。在这里，将 fn 置为打印 EPT 页表项的函数，如下打印格式与 GPT 相同。

gpt - dump/ept - dump.txt

```
NEXT_LVL_HPA(EPTP) =   000000000000000000011101011101100010010 1   +   64 * 0

0   : PGD 000000000000 000000000000000000010001000011000100101 0111 000100000111
NEXT_LVL_HPA(PGD)   =   000000000000000000010001000011000100101 0111   +   64 * 0

0   : PUD 000000000000 000000000000000000010000000101000101001 00100 000100000111
NEXT_LVL_HPA(PUD)   =   000000000000000000010000000101000101001 00100   +   64 * 469

469: PMD 000000000000 000000000000000000010001000001110000111 11111 000100000111
NEXT_LVL_HPA(PMD)   =   000000000000000000010001000001110000111 11111   +   64 * 163

163: PTE 000000000000 000000000000000000011010110111101001101 1 111101110111
```

由于本实验没有关闭宿主机上的大页，KVM 在查询到最后一级页表时做了两种处理：如果遍历到大页，则调用 print_huge_pte 函数打印最后获取 HPA 的过程；否则调用 print_pte 函数。具体的打印代码不再赘述，下面只展示最后如何使用代码得到 HPA。

linux - 4.19.0/arch/x86/kvm/mmu.c

```
unsigned long gpa_from_guest, pxx_idx[PT64_ROOT_4LEVEL];

// 大页的情况，仅考虑 2MB 的大页
static inline u64 print_huge_pte(u64 ent) {
        // 最终宿主机物理页的 HPA
```

```
    u64 page_hpa = ent & PT64_DIR_BASE_ADDR_MASK,
        // 获取页偏移的 mask
        offset_mask = PT64_LVL_OFFSET_MASK(2) | (PAGE_SIZE - 1),
        // 物理页 HPA 加上页偏移,最终调用__va()得到 HVA
        *ptr = (u64 *)(__va(page_hpa + (gpa_from_guest & offset_mask)));

    // 最终读取 ptr 处的数据,为 0xdeadbeef
    pr_info("Value at HPA: %p\n", (void *) *ptr);
}

// 普通 4KB 页的情况
static inline u64 print_pte(u64 ent) {
        // 最终宿主机物理页的 HPA
    u64 page_hpa = ent & PT64_BASE_ADDR_MASK,
        // 获取页偏移的 mask
        offset_mask = PAGE_SIZE - 1,
        // 物理页 HPA 加上页偏移,最终调用__va()得到 HVA
        *ptr = (u64 *)(__va(page_hpa + (gpa_from_guest & offset_mask)));

    // 最终读取 ptr 处的数据,为 0xdeadbeef
    pr_info("Value at HPA: %p\n", (void *) *ptr);
}
```

对于 2MB 大页的情况，最后一级页表项是 PDE，这类页表项的第 **51:21** 位表示大页的起始物理地址，这里使用 PT64_DIR_BASE_ADDR_MASK 宏从 ent 页表项中获取大页的起始物理地址。接下来，使用 PT64_LVL_OFFSET_MASK(2) | (PAGE_SIZE-1)获取 GPA 中的大页偏移的部分，即 20:0 位。最终，结合大页起始地址和大页偏移得到 HPA，最后调用__va 函数获取对应的 HVA，并解引用该 HVA，读取到数据 **0xdeadbeef**，符合预期。

对于普通 4KB 页的情况，PTE 中的第 **51:12** 位表示其指向的物理页的起始地址，使用 PT64_BASE_ADDR_MASK 宏从 PTE 中获取页的物理地址，再使用 PAGE_SIZE-1 从 GPA 中获取页偏移，即 11:0 位。最后结合页的物理地址和页偏移得到 GPA，最后调用__va 函数得到 HVA，并解引用该 HVA，读取到数据 **0xdeadbeef**，符合预期。

综上所述，客户机内核模块在一个 GVA 处写入了 **0xdeadbeef**，读取客户机页表得到 GVA 对应的 GPA，通过超级调用传递 GPA 到 KVM 模块，KVM 读取 EPT 将 GPA 翻译成 HPA，最后通过__va 函数找到 HPA 对应的 HVA，并读取到 **0xdeadbeef**，表明 HPA 处确实存储了 GVA 处的数据，地址翻译成功。在翻译过程中，实验代码打印了地址翻译所涉及的页表项、GPA、HPA 等，环环相扣。ept-dump.txt 文件存储了完整的输出信息，供读者查看。

3.4　GiantVM 内存虚拟化

第 2 章结尾介绍了“多虚一”的虚拟机监控器 GiantVM，基于 QEMU/KVM 的 Type Ⅱ 开源虚拟机监控器运行在多个物理节点上，为操作系统提供了一个透明的“虚拟”物理硬

件，而这些“虚拟”物理硬件的后台实现是多个物理节点上的物理资源。这种分布式的虚拟机监控器既可以聚合海量的计算资源，为资源要求高的工作负载（如机器学习、大数据分析）提供便利，也可以覆盖分布式系统的复杂性，程序员可以将在单机上运行的程序无修改地运行在一个分布式系统上，即提供一个 SSI，如本章开头多机上的“虚拟”物理地址空间所述。

分布式框架（如 Spark、MapReduce）往往有复杂的编程模型，程序员想要让自己的程序运行在分布式系统上，则必须要根据这些分布式框架提供的接口改写旧的代码；而巨型虚拟机 GiantVM 可以运行任何一个普通的操作系统，给程序员提供与单机环境上完全相同的接口，无须修改原有的代码即可以将程序运行在海量资源之上，甚至原来难以运行在分布式集群上的应用也可以借助巨型虚拟机运行在分布式集群上。

在一个运行在 GiantVM 的客户机看来，它拥有大量的 CPU 和物理内存。经过测试，GiantVM 可以运行具有 512 个 vCPU 和 2TB 内存的 sv6 操作系统[①]。第 2 章已经介绍了 CPU 的虚拟化以及跨节点中断转发的实现。作为 QEMU/KVM 内存虚拟化原理的拓展，本章介绍分布式共享内存的原理以及内存虚拟化在 GiantVM 中的实现。

3.4.1 分布式共享内存

如 3.1 节所述，多机上的“虚拟”物理内存的后台实现是多个物理节点，每个节点都拥有一定数量的物理内存，将这些物理内存通过虚拟化的方式聚合起来，建立一个虚拟“物理内存”的抽象层，就可以供操作系统无修改地运行。接下来的问题是：以什么样的方式建立该抽象层呢？容易想到，应该用地址翻译建立抽象层，下文讨论如何完成地址的管理。

从熟悉的 CPU 缓存架构出发：每个 CPU 核都有一个私有的一级/二级缓存（L1/L2 cache，L1 cache 大小一般为 32KB，L2 cache 大小一般为 256KB），它们保存着主存（内存）中数据的副本。每次访问主存之后，需要将访问地址周围**缓存行大小**（一般为 64 字节）的数据复制到缓存中，缓存行是主存和缓存之间数据交换的单位。为了保证每个 CPU 核的缓存中副本的正确性，有一套 **MESI** 缓存一致性协议控制每个缓存行的状态，包括已修改的（Modified，M）、独占的（Exclusive，E）、共享的（Shared，S）、无效的（Invalid，I）四种状态。CPU 对缓存行的读写操作都将触发缓存行状态的改变，例如 CPU 读取一个无效的缓存行，则首先将该缓存行对应数据的最新版本写入主存，再将该数据拷贝到该缓存行中，最后将该缓存行的状态置为共享。这样每个 CPU 核心都能够读到该数据的最新副本，从而避免读取到旧的数据产生错误。缓存架构在 Intel SDM 中有详细介绍，有兴趣的读者可以查阅。

回想多台机器上的“虚拟”物理地址空间，对它的管理类似于 CPU 缓存架构：将每个物理节点类比为一个 CPU 核心，而“虚拟”物理地址空间视作主存，那么每个物理节点的内存中均保存着“虚拟”物理地址空间中数据的一个副本。为了减少网络访问，每个机器的内存和全局的“虚拟”物理内存之间交换数据的单位是页，大小为 4KB，而非 64 字节的缓存行，

① 代码仓库地址：https://github.com/aclements/sv6，sv6 是一个实验性质的操作系统。

每个页都有一个类似于 MESI 的状态。于是，如果一个访存指令访问一个**“虚拟”物理地址空间**中的地址，那么首先在执行这条指令的机器的内存中查看是否保存了一份该数据的副本，如果没有此副本，则向一个全局的控制者询问：这份数据保存在哪里？于是可以取得该数据的最新副本，并复制到本地内存。这样访问“虚拟”物理地址空间中地址的指令就可以访问整个分布式集群中所有节点的物理内存，虽然需要经过网络，但还是实现了内存的聚合。

可能有读者会产生疑问：如果给每台机器划分出一块“虚拟”物理地址空间的范围供其专门管理，只要访问的“虚拟”物理地址落在某个机器所管辖的范围内，那么是否该地址对应数据的副本就在这台机器上？例如，如果要实现一个 16G 的物理地址空间，一共有 4 个节点，那么节点 1 负责前 1/4 的地址空间，节点 2 负责第二个 1/4 的地址空间，以此类推。这种实现方式本质上还是需要将远程节点的数据缓存到本地，并维护每个缓存页的状态，否则会产生大量的网络访问，例如，如果节点 1 频繁访问节点 2 所管辖的内存范围，那么节点 1 就需要通过网络频繁访问节点 2 的内存，造成性能问题。

GiantVM 的构想与多年前 IVY 这一 DSM（Distributed Shared Memory，分布式共享内存）系统设计者 Kai Li[①] 的理念不谋而合：如图 3-18 所示，为了维护数据一致性，IVY 定义了已修改、共享、无效三种状态，每个页都处于三种状态之一。无效状态表示当前页的内容无效，此时该页不可读写，若进行读写则产生缺页异常，进入 IVY 进行状态迁移；共享状态表示当前页可能被多台机器共享，此时该页只读，若进行写入则触发状态迁移；已修改状态表示当前页被一台机器独占，此时该页可由这台机器读写。在状态迁移的处理上，IVY 采取了与基于目录的一致性协议类似的做法，引入了**管理者（Manager）**的概念，相当于缓存一致性算法中的目录（Directory），由管理者跟踪每个页的**持有者（Owner）**。对无效页进行读取时，IVY 系统通过查询管理者找到页的持有者，若是已修改页，则该已修改页转到共享状态，否则不变，然后将请求者（该无效页所在的机器）加入持有者的**复制集（Copyset）**，最后将该页的拷贝返回给请求者，该页即可以从无效状态转移到共享状态。类似地，对无效或共享页进行写入时，IVY 系统也是通过查询管理者找到页的持有者，获取拷贝，不过此时还需要获取其复制集，取得拷贝后完成写入，最后向复制集中的所有机器发送无效消息，令它们上面的所有对应页进入无效状态，即可以完成写入，使写入者转移到已修改状态。

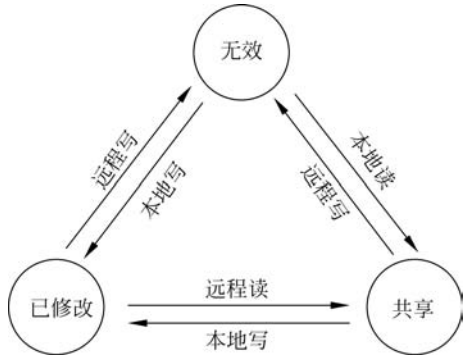


图 3-18　MSI 协议状态变迁图

当客户机用“虚拟”物理地址进行内存访问时，就可以复用 IVY 系统的 MSI（Modified、

① http://css.csail.mit.edu/6.824/2014/papers/li-dsm.pdf 中介绍了 Kai Li 设计的分布式共享内存实现。

Shared、Invalid，已修改的、共享的、无效的）页状态管理机制，使用“虚拟”内存地址查找真实数据在分布式集群中的位置。IVY 向上层操作系统暴露与实际内存完全相同的语义，即随机访问和按字节寻址。有了 DSM 模型，就可以讨论如何在 QEMU/KVM 中实现该模型，即实现 GiantVM 的内存虚拟化。

3.4.2 GiantVM 中的 DSM 架构

下文将基于 QEMU/KVM 二次开发的分布式 QEMU 称为 **dQEMU**（distributed QEMU，分布式 QEMU），当它单独部署到一台物理机上时，就退化成为普通的 QEMU；而当部署到多台物理机上时，则每台物理机上都会启动一个 dQEMU 实例（即 dQEMU 进程），dQEMU 与其所在机器的 KVM 模块协同工作，由多个 dQEMU 实例通过网络共同构成一个分布式的虚拟机监控器。GiantVM 的实现既有基于 RDMA（Remote Direct Memory Access，远程内存访问）的版本，也有基于 TCP（Transmission Control Protocol，传输控制协议）的版本。RDMA 由于其绕过了远端操作系统，性能比 TCP 更高，但需要特殊的硬件。

首先，在 dQEMU 内存参数方面，假设“虚拟”物理内存的大小是 2TB，在启动 dQEMU 进程时依然采用参数-m 2T，于是在每个节点上运行的 vCPU 依然可以访问 2TB 的“虚拟”物理内存。此时 dQEMU 进程仅仅使用 mmap 在宿主机上分配的 2TB 虚拟内存，并未分配实际的物理内存，未建立 GPT 和 EPT 的相关表项。而具体访问的数据保存在哪个节点上由 KVM 进行管理，即在 KVM 中仿照 IVY 系统实现 DSM，这是由于 KVM 可以管理 EPT，EPT 管理了“虚拟”物理内存。其次，为了让客户机操作系统对巨型虚拟机的实际物理拓扑有一定认识，GiantVM 将物理机抽象成 NUMA 节点的形式呈现给客户机（通过 dQEMU 的-numa 参数），整个巨型虚拟机将被理解为一台巨大的 NUMA 机器，其中每个 NUMA 节点对应于一台物理机器，这样客户机操作系统的进程调度和迁移、内存管理等都会对巨型虚拟机的物理拓扑有所参考。

在实现的内存模型方面，由于 x86 硬件给操作系统提供的内存模型是 x86-TSO（x86 Total Store Order，x86 全存储序），因此 GiantVM 实现的内存模型的一致性不能弱于 x86-TSO，否则一般的操作系统无法正确运行。而 IVY 提供了顺序一致性的内存模型强于 TSO，故可以将 IVY 系统整合进 KVM，即可实现一个具有顺序一致性保证的“虚拟”物理内存平台。限于篇幅，此处暂不考虑更宽松的内存一致性模型。图 3-19 展示了在 GiantVM 中处理客户机读一个无效页的过程。整个过程的第 1 步是通过页表权限的控制，令客户机在执行内存读取操作时产生异常，退出到 KVM。随后，根据内存同步协议，需要先向管理者请求，然后管理者将请求转发给持有者，即第 2、3 步。这里，KVM 专门开辟了一个内核线程作为服务端，用于接收其他节点发来的消息。持有者收到消息后，要访问客户机的内存，读取该页的内容，即第 4、5 步，随后才能将其返回给请求者。最后，请求者获得该页的副本后，还要先写入客户机的内存，即第 8、9 步，并修改页表权限、将页转到共享状态，然后才能返回客户机模式。

下文将介绍在 KVM 中如何实现一个类似于 IVY 系统的内存管理系统，可以和前文

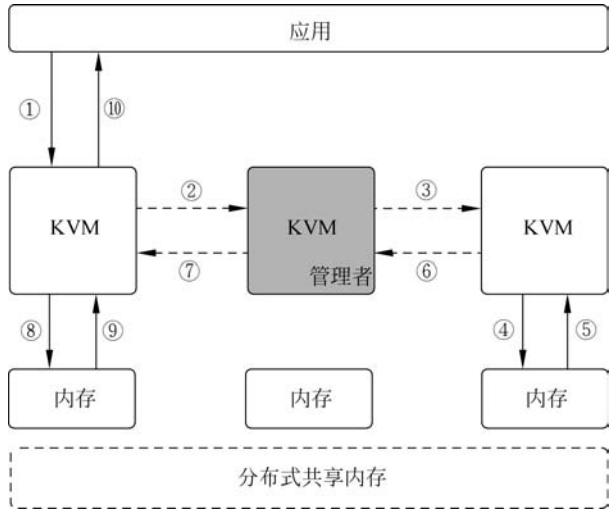


图 3-19　IVY 在 GiantVM 中的实现

KVM 内存虚拟化的代码相互照应，作为一个拓展。GiantVM 内存虚拟化的实现基于 **EPT** 的管理，由于 EPT 存储了“虚拟”物理内存页的一切信息，且与客户机页表完全解耦。

3.4.3　GiantVM 中 DSM 的实现

本节简要介绍如何在 KVM 中实现上文提出的 10 个处理步骤，可能涉及 vCPU 线程以及其他处理线程，均在内核态运行，下面分步介绍。

1. 页表权限的设置

可以看到，整个流程之所以被触发，就是因为页的权限设置，可以说它是内存同步协议成立的重要原因，在整个系统中的地位不言而喻。由于 KVM 同时支持影子页表和二级页表（即 EPT）两种内存虚拟化模式，权限设置对应地也有两种，即设置影子页表或 EPT 中页表项的权限。实际运用中，影子页表模式基本上已被淘汰，因此仅考虑 EPT 模式的支持。

首先考察 KVM 本身对页表的管理。在 KVM 中，客户机触发缺页异常（影子页表模式）或 EPT Violation（EPT 模式）后，最终会来到 kvm_mmu_page_fault 函数，然后根据模式的不同，通过函数指针调用不同的处理函数。EPT 模式对应的函数为 tdp_page_fault 函数，其中最重要的两个步骤分别是调用 try_async_pf 函数和调用 __direct_map 函数。tdp_page_fault 函数的参数是 GPA，前一个步骤所做的工作就是找到它对应的 HVA，然后根据 HVA 找到对应的 PFN（Physical Frame Number，宿主机物理页框号）。后一个步骤所做的工作则是根据 GFN（Guest Frame Number，客户机页框号）和 PFN 建立页表，这里就是建立 EPT 页表。最终的权限设置在 set_spte 函数中进行。

GiantVM 在上述两个步骤之间增加了一个步骤，即调用 kvm_dsm_page_fault 函数执行内存同步协议，这一步的返回值是 IVY 协议想为新建立的页表项设置的权限。EPT 表

项基本上有三个权限位，即 R、W、X，分别是 EPT 表项的第 0～2 位。对于已修改页，三个权限位都设置为 1，对应的返回值为 ACC_ALL；对于共享页，将 R、X 位设置为 1，对应的返回值为 ACC_EXEC_MASK | ACC_USER_MASK；对于无效页，三个位都设置为 0。取得返回值后，传入 __direct_map 函数，最后一路传到 set_spte 函数，就实现了对页表项权限位的控制。下面代码展示了页表项权限位的控制流程。

giantvm - linux/arch/x86/kvm/mmu.c | dsm.c

```
vCPU thread -> handle_ept_violation -> kvm_mmu_page_fault ->
tdp_page_fault {                              // mmu.c
    if (try_async_pf(.., &pfn, ..))
        return 0;
    dsm_access = kvm_dsm_vcpu_acquire_page(vcpu, &slot, gfn, write);
    r = __direct_map(.., dsm_access);
}

kvm_dsm_vcpu_acquire_page ->                  // 是 dsm 模块向 mmu 模块提供的一个接口，见 dsm.c
    __kvm_dsm_acquire_page ->
        kvm_dsm_page_fault ->
            ivy_kvm_dsm_page_fault            // 根据 IVY 协议得出页状态

__direct_map(dsm_access) ->
    mmu_set_spte(pte_access) ->
        set_spte(pte_access) ->
            mmu_spte_update(sptep, new_spte) ->
                mmu_spte_set(sptep, new_spte) ->
                    __set_spte(sptep, new_spte) ->
                        WRITE_ONCE( * sptep, spte);   // 最终更新 EPT 表项
```

上面只介绍了对于权限的升级，相对应的降级操作则是在服务端内核线程进行的，例如收到无效化请求后，就要将自身的状态切换到无效。这里的操作更简单，可以直接利用 mmu_spte_update 函数修改页表项，不过此处使用的是 KVM 里进一步封装的函数。详细调用过程如下。

giantvm - linux/arch/x86/kvm/dsm.c | ivy.c

```
kvm_vm_ioctl_dsm ->                 // dsm.c
    kvm_dsm_init ->
        thread = kthread_run(kvm_dsm_threadfn, (void * )kvm,          // 启动主处理线程
                             "kvm - dsm/ % d", kvm -> arch.dsm_id);

// kvm_dsm_threadfn 在接收到一个请求后，启动 NDSM_CONN_THREADS 个 req 处理线程
kvm_dsm_threadfn {
    while (1) {
        ret = network_ops.accept(listen_sock, &accept_sock, 0);       // 接收请求
        for (i = 0; i < NDSM_CONN_THREADS; i++) {
            thread = kthread_run(kvm_dsm_handle_req,
                (void * )conn, "dsm - conn/ % d: % d", kvm -> arch.dsm_id, count++);
```

```
        }
    }
}

kvm_dsm_handle_req -> ivy_kvm_dsm_handle_req {                    // ivy.c
    while (1) {
        switch (req.req_type) {
        case DSM_REQ_INVALIDATE:
            ret = dsm_handle_invalidate_req(kvm, conn_sock,
                    memslot, slot, &req, &retry, vfn, page, &tx_add);
        case DSM_REQ_WRITE:
            ret = dsm_handle_write_req(kvm, conn_sock, memslot, slot, &req,
                    &retry, vfn, page, &tx_add);
        case DSM_REQ_READ:
            ret = dsm_handle_read_req(kvm, conn_sock, memslot, slot, &req,
                    &retry, vfn, page, &tx_add);
        }
    }
}
```

2. 客户机内存的访问

上文主要围绕图 3-19 中的步骤 1 展开，下面考察步骤 4、5 和 8、9，即对客户机内存的读写访问的实现，相对而言这要简单得多。在 KVM 中，原本就有读写客户机内存的功能，一些半虚拟化特性（如时钟等）都会用到，它们最终分别是由 __kvm_read_guest_page 和 __kvm_write_guest_page 这两个函数实现的。因此，对于步骤 8、9，KVM 可以直接调用这两个函数对客户机内存进行读写。但是，对于步骤 4、5，情况则有所不同。

步骤 8、9 实际上位于上文介绍的 kvm_dsm_page_fault 函数中（它又在 tdp_page_fault 函数中），而步骤 4、5 则是在专门开辟的服务端内核线程中执行的。这两者最大的区别在于，前者位于 QEMU 进程的 vCPU 线程中，而后者作为内核线程，没有对应的用户态程序。KVM 在实现读写客户机内存的功能时，是通过 GPA 找到对应的 HVA，然后使用 Linux 内核提供的 __copy_from_user 和 __copy_to_user 调用，从用户地址空间读取或向其写入。对于前者，用户地址空间就是 QEMU 进程的地址空间，因此操作能够成功。对于后者，不存在对应的用户地址空间，因此调用会失败。事实上，Linux 内提供了为内核线程临时挂载用户地址空间的功能。服务端内核线程进行客户机内存读写操作时，首先通过 use_mm 函数挂载 KVM 所在的 QEMU 进程的地址空间，然后便可以进行读写操作，最后用 unuse_mm 函数取消挂载，恢复原状。这样就解决了服务端内核线程中无法对客户机进行内存读写的问题，调用链如下。

giantvm - linux/arch/x86/kvm/ivy.c

```
dsm_handle_read_req/dsm_handle_write_req ->
    kvm_read_guest_page_nonlocal
    {
        use_mm(kvm->mm);
```

```
        ret = __kvm_read_guest_page(slot, gfn, data, offset, len)
        {
            addr = gfn_to_hva_memslot_prot(slot, gfn, NULL);
            r = __copy_from_user(data, (void __user *)addr + offset, len);
        }
        unuse_mm(kvm->mm);
    }
```

3. 页面状态的维护

通过上文介绍，读者可以得知分布式共享内存如何进行必要的操作，如页表权限管理和客户机内存访问。构成分布式共享内存的另一个核心要素就是与每个页相关联的 MSI 状态，可以说内存同步协议就是围绕页的状态转移展开的。显而易见，节点之间发送的同步消息，例如读取某个页或无效化某个页的请求，必然要指明请求的是哪个页，否则无从确定此同步操作的对象。由于各个节点的 dQEMU 进程不可能有共同的虚拟地址，它们一定只能使用 GFN 指定操作对象。由此自然得出的一个结论就是，页的 MSI 状态以及复制集等信息也应该与 GFN 绑定。

在 QEMU/KVM 中，客户机的内存本质是在 QEMU 中申请的一段内存，QEMU 可以通过 HVA 直接访问它。在 QEMU 中，通过使用由 MemoryRegion 对象构成的树来管理内存，QEMU 将内存注册到 KVM 时还原成一维的内存区间进行注册，每个区间具有起始地址（包括 HVA 地址和 GPA 地址）和大小。在 KVM 中，上述内存区间由 kvm_memory_slot 表示，正如 3.3.3 节介绍。一种解决方案是，向该数据结构中添加一项 struct kvm_dsm_info * gfn_dsm_state，用于维护分布式共享内存中的页面信息。在服务端内核线程收到消息后，根据 GFN 能很容易地找到对应的内存插槽并进一步检查或修改页面的 MSI 状态等。

然而，在上述原型系统中，会偶尔出现内存状态不一致的现象，例如本该被标为无效的页，客户机却没有经过同步就读取到了它的内容。经过进一步的排查和调研，这是因为对于同一个 GFN 也就是同一个页，系统实际上维护了两份 kvm_dsm_info，其中一份被标记为无效时，另一份可能仍处于共享状态因此可以被读取。这是因为两个不同的 GFN 对应的是同一块宿主机的内存，QEMU 在注册时为同一段宿主机虚拟地址注册了两次，分别使用了不同的 GPA。这一般是为了模拟一些硬件在 1MB 以内的低地址有一个为兼容而存在的传统 MMIO 区域，在高地址又有一个 MMIO 区域，两个 MMIO 区域能访问到的内容是相同的。

简而言之，在 QEMU/KVM 中，一个 **HVA** 可能对应**多个 GPA**，而每个 HVA 到 GPA 的映射都对应一个 kvm_memory_slot，从而维护一份页面信息，而 GPA 到 HVA 的映射是一对一的。要解决这些问题，办法就是将页面信息与 HVA（即 QEMU 中的地址）绑定，而不是与 GFN 绑定。为此，①GiantVM 引入了 kvm_dsm_memory_slot 数据结构（以下简称 hvaslot），用于维护 QEMU 申请的 HVA 内存区块的信息，页面状态等信息转为放入 hvaslot 中维护，这样就可以避免同一块内存拥有多份页面信息。②仍然使用 **GFN** 作为节

点间传递的信息。当客户机发生缺页异常或服务端内核线程收到消息时，可以根据待处理的页的GFN，利用内存插槽中维护的**GFN到HVA**的映射，找到对应的hvaslot，进而读取或修改页面的MSI状态等。③另外，GiantVM还需要在hvaslot中维护从**HVA到GFN**的反向映射，这是KVM自身没有维护的。当GiantVM需要根据内存同步协议修改一个页的权限时，不能只修改同步消息中GFN对应的页表项，应该由该GFN找到对应的HVA，再根据反向映射找到所有关联的GFN，修改所有这些GFN对应的页表项的权限。经过上述重构后，此前随机出现的内存不一致现象得到了消除，客户机能长时间稳定运行。

在源码实现里，kvm_dsm_memory_slot由kvm_dsm_memslots管理，存储在struct kvm_arch中，不同于kvm_memslots直接存储在struct kvm中，数据结构之间的关系见下面的代码。

giantvm - linux/arch/x86/include/asm/kvm_host.h

```
struct kvm -> struct kvm_arch arch;                    // include/linux/kvm_host.h
struct kvm_arch {
    struct kvm_dsm_memslots * dsm_hvaslots;
}

struct kvm_dsm_memslots {
    struct kvm_dsm_memory_slot memslots[KVM_MEM_SLOTS_NUM];     // hvaslot 数组
    int used_slots;
};

struct kvm_dsm_memory_slot {
    hfn_t base_vfn;                                    // HVA 起始地址
    unsigned long npages;
    struct kvm_dsm_info * vfn_dsm_state;               // kvm_dsm_info 数组
};

struct kvm_dsm_info {
    // HVA 页状态，包括 DSM_INVALID、DSM_SHARED、DSM_MODIFIED、DSM_OWNER 等
    unsigned state;
    DECLARE_BITMAP(copyset, DSM_MAX_INSTANCES);        // 复制集位图
#ifdef KVM_DSM_DIFF
    struct {...} diff;                                 // 压缩优化相关状态，见下文
#endif
};
```

kvm_dsm_memory_slot的添加流程与kvm_memory_slot相同，和3.3.4节所述类似，从QEMU调用ioctl开始，到添加kvm_memory_slot的函数kvm_arch_create_memslot→kvm_dsm_register_memslot_hva为止，有兴趣的读者可以查阅源码。kvm_dsm_memory_slot是DSM协议实现的关键数据结构，在代码中经常出现。

4. 网络通信

至此已经基本介绍完分布式共享内存在KVM中的实现以及遇到的问题，不过还有网

络通信的实现，即图 3-19 中的 2、3、6 和 7 步，尚未展开说明。

为了减少在网络中需要传输的数据，GiantVM 进行了压缩优化。提供编码和解码两个原语(dsm_encode_diff/dsm_decode_diff 函数)，需要在网络中传输页面时，编码函数得到新旧页数据的差，并进行编码；而解码函数对编码函数得到的差进行解码，与旧页数据进行合并，得到新页数据。压缩优化的实质在于当节点 1 向节点 2 传输数据时，如果节点 2 已经保存了节点 1 所传输数据的旧版本，那么只需要传输两者**数据之差**。下面用一次读取请求作为例子，介绍压缩优化的使用，代码如下。

giantvm - linux/arch/x86/kvm/ivy.c

```
struct kvm_network_ops network_ops;      // 网络请求函数集，见 arch/x86/kvm/dsm - util.c

// vCPU 线程(发起读取请求)
int ivy_kvm_dsm_page_fault(...)          // 调用链见 3.1 节 {
    char * page = NULL;
    page = kmalloc(PAGE_SIZE, GFP_KERNEL);

    if (write) {
        ... // 处理写引起的缺页异常
    } else {
        // 省略 DSM 协议实现
        ret = resp_len = kvm_dsm_fetch(kvm,owner,false,&req, page, &resp) ->
        {
            ret = network_ops.send(...);
            ...
            ret = network_ops.receive( * conn_sock, page,...);
        } // kvm_dsm_fetch
    }

    dsm_decode_diff(page, resp_len, memslot, gfn);
    __kvm_write_guest_page(memslot, gfn, page, 0, PAGE_SIZE);
}

// 服务端内核线程(处理读取请求)
static int dsm_handle_read_req(...char * page) {
    ...
    kvm_read_guest_page_nonlocal(kvm, memslot, req -> gfn, page, 0, PAGE_SIZE);
    ...// 省略 DSM 协议的实现
    if (is_owner)
        length = dsm_encode_diff(slot, vfn, req -> msg_sender, page, memslot,
                req -> gfn, req -> version);
    ret = network_ops.send(conn_sock, page, length, 0, tx_add);
}
```

在此实例中，一个节点上的 KVM 发生了 EPT 缺页异常，最终调用 IVY 协议的缺页处理函数 ivy_kvm_dsm_page_fault。它首先分配一个 PAGE_SIZE 大小的页作为读取结果的存放页(这是由于 DSM 中网络传输的单位是 4KB 页，而非 64 字节大小的缓存行)，page 指

向该页。接下来，向该页的持有者发起读取请求，调用 kvm_dsm_fetch 函数，最后通过网络接收到服务端内核线程返回的编码后的数据。于是，page 被压缩后的页内容填充，使用 dsm_decode_diff 函数进行解码，最后__ kvm_write_guest_page 函数写入客户机物理内存。

服务端内核线程接收到客户端的读请求后，首先从客户机物理内存中读取该页，并调用 dsm_encode_diff 函数进行编码，再使用 send 接口向网络发送该页，就能减小网络开销。

另一种减小网络开销的方式是使用**绕过内核**的网络传输。具体来说，目前一共有三种 RDMA 解决方案，即 Infiniband(无限带宽技术)、RoCE(RDMA over Converged Ethernet，基于统合式以太网的 RDMA)和 iWARP(internet Wide-Area RDMA Protocol，互联网广域 RDMA 协议)。它们共同的特点是低延迟、高带宽，并且支持用户态协议栈，即应用程序可以不经过内核而直接操控网卡发送数据包，因此可以带来很大的性能提升。巨型虚拟机使用 Infiniband 解决方案。使用 RDMA 的 network_ops 如下。

giantvm - linux/arch/x86/kvm/dsm.c

```
struct kvm_network_ops { // arch/x86/kvm/dsm - util.h
    int ( * send)(kconnection_t * , const char * , size_t, unsigned long,
            const tx_add_t * );
    int ( * receive)(kconnection_t * , char * , unsigned long, tx_add_t * );
    int ( * connect)(const char * , const char * , kconnection_t ** );
    int ( * listen)(const char * , const char * , kconnection_t ** );
    int ( * accept)(kconnection_t * , kconnection_t ** , unsigned long);
    int ( * release)(kconnection_t * );
};

//使用< rdma/ib_verb.h >/< rdma/rdma_cm.h >
static int kvm_dsm_init() {
#ifdef USE_KRDMA_NETWORK
    network_ops.send = krdma_send;
    network_ops.receive = krdma_receive;
    network_ops.connect = krdma_connect;
    network_ops.listen = krdma_listen;
    network_ops.accept = krdma_accept;
    network_ops.release = krdma_release;
#endif
}
```

5. 伪共享问题

显而易见，巨型虚拟机内存虚拟化系统的性能瓶颈在于网络开销。在 DSM 系统中，**伪共享**会造成巨大的网络开销，严重降低 GiantVM 的性能。伪共享是指两个没有联系的变量处在同一个页上，假设是变量 X1 和 X2。节点 1 不断地写入变量 X1，导致在节点 2 上，X1 所在的页被置为无效；而当节点 1 频繁写入 X1 变量时，节点 2 频繁地读取 X2，而它发现 X2 所在的页被设为无效，就需要通过网络从节点 1 获取该页的最新副本，开销很大。这种情况在巨型虚拟机中十分常见，一种解决方法是将互不相关的变量分配在不同的页上，

通过内核编译选项实现；另一种解决方案是将经常访问共享页面的 vCPU 调度到同一个节点上，这样就不需要经过网络。

本章小结

本章首先介绍了内存虚拟化的抽象原理，说明了内存虚拟化的核心是维护地址的映射；再介绍了现实世界中内存虚拟化的实现方式，包括影子页表与扩展页表，并介绍了敏捷页表作为扩展；接下来对 QEMU/KVM 中的内存虚拟化实现进行深入介绍，最后一小节作为对内存虚拟化原理以及 QEMU/KVM 源码的拓展，介绍了巨型虚拟机“多虚一”系统的实现。源码章节中的实验部分有助于读者更加直观地理解源码实现，而源码实现是较难理解的部分。内存作为云环境的重要计算资源，对程序性能有着举足轻重的影响。内存虚拟化作为云环境中内存资源的基石，值得进一步做性能优化，等待读者去探索。