第 3 章

计算虚拟化

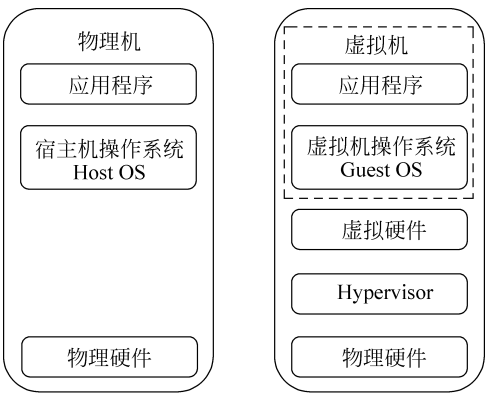
第 2 章已经对数据中心资源虚拟化的三大核心组件(计算虚拟化、存储虚拟化及网络虚拟化)进行了初步阐述,本章将聚焦于计算虚拟化部分。计算虚拟化作为虚拟化技术的一种重要形式,其抽象粒度覆盖整个服务器层面。近年来,随着处理器技术的迅速发展与性能的显著提升,虚拟化技术的成熟时机已然到来。特别是硬件虚拟化技术的诞生,如 Intel VT 和 AMD SVM 技术,极大地拓宽了计算虚拟化的应用领域。

3.1 节将对计算虚拟化的基本概念进行简要概述,帮助读者明确计算虚拟化所要解决的核心问题。3.2～3.5 节解析计算虚拟化的原理与实现方式,涵盖 CPU 虚拟化、内存虚拟化、I/O 虚拟化以及 GPU 虚拟化,并探讨其不同的实现策略。3.6 节介绍华为 DCS 在计算虚拟化方面的实际应用案例,帮助读者更深入地理解和掌握计算虚拟化的相关知识。

3.1 计算虚拟化概述

计算虚拟化作为数据中心虚拟化的重要组成部分,是一种通过虚拟化技术将物理计算资源抽象成虚拟资源的技术。它将一台物理机的计算资源(如 CPU、内存、I/O 设备等)进行虚拟化,然后动态地分配给多个虚拟机使用。通过这种方式,计算虚拟化实现了计算资源的灵活调度、高效利用和统一管理。

为了更好地理解计算虚拟化,可以将虚拟机与物理机层次体系结构进行对比。如图 3-1(a)所示,在物理机中,操作系统直接运行在硬件之上,享有对硬件资源的直接访问权。而在虚拟化环境中,每个虚拟机都运行在自己的虚拟化环境中,通过 Hypervisor 进行管理和调度,如图 3-1(b)所示。Hypervisor 是计算虚拟化的核心组件,它负责在物理硬件和虚拟机之间建立一层抽象层,使得虚拟机能够像物理机一样运行,但同时又能够共享和隔离硬件资源。



(a) 物理机层次结构　　(b) 虚拟机层次结构

图 3-1　物理机和虚拟机层次结构对比

Hypervisor 的作用不仅限于资源管理和调度,还负责提供虚拟机之间的通信机制、安全隔离措施以及性能优化等功能。通过 Hypervisor,可以实现对虚拟机的动态创建、迁移、删除等操作,从而根据实际需求灵活地调整物理资源分配。此外,Hypervisor 还能够

提供对虚拟机的监控和统计功能，帮助管理员更好地了解虚拟机的运行状态和性能表现。

现代计算机最核心的三类资源是CPU、内存和I/O设备。如果计算虚拟化要构建出可以运行的虚拟机，CPU虚拟化、内存虚拟化和I/O虚拟化是必要的。此外，随着视频图像类应用得到了越来越普遍的使用，图形处理器(Graphic Processing Unit，GPU)凭借出色的并行处理能力，被广泛集成到计算机系统中。因此，GPU虚拟化对提高虚拟机应用程序的运行性能和用户体验至关重要。综合来看，计算虚拟化需要应对的问题以及实现的功能简要概括如下。

(1) CPU虚拟化。CPU虚拟化主要解决的是如何在物理CPU上创建多个虚拟CPU，并让它们能够像物理CPU一样运行各种操作系统和应用程序的问题。它通过指令集模拟、中断和异常处理等技术，实现了虚拟机对物理CPU的透明访问。CPU虚拟化不仅提高了CPU资源的利用率，还增强了系统的灵活性和可扩展性。同时它还需要确保虚拟机之间的隔离性和安全性，防止相互干扰和攻击。

(2) 内存虚拟化。内存虚拟化关注的是如何在物理内存和虚拟机之间建立映射关系，使得虚拟机能够像访问物理内存一样访问虚拟内存。内存虚拟化通过内存管理单元(MMU)和页表等机制，实现了内存的共享、隔离和保护。它提高了内存的利用率，减少了内存的浪费。同时，内存虚拟化还需要保证虚拟机内存访问的性能和稳定性，确保虚拟机能够高效、稳定的运行。

(3) I/O虚拟化。I/O虚拟化主要解决的是虚拟机与外部设备之间的通信问题。它使得虚拟机能够像物理机一样访问存储设备、网络设备等，实现了I/O操作的透明性和隔离性。I/O虚拟化通过设备模拟、直接I/O访问和I/O透传等方式，提高了I/O操作的性能和效率。同时，它还需要考虑I/O操作的安全性和可靠性，确保虚拟机与外部设备之间的通信稳定可靠。

(4) GPU虚拟化。GPU虚拟化主要解决的是物理GPU资源分配不灵活、多个虚拟机或应用之间的GPU资源争抢、高昂的硬件成本等问题。GPU虚拟化通过将物理GPU资源分割成多个虚拟GPU(vGPU)，并分配给不同的虚拟机或应用使用，提高资源利用率和灵活性，同时降低硬件成本。

计算虚拟化通过对CPU虚拟化、内存虚拟化、I/O虚拟化和GPU虚拟化等技术的综合运用，实现计算资源的共享、隔离和高效利用。下面将对上述计算虚拟化的问题和内容进行展开，概述计算虚拟化(包括CPU、内存、I/O和GPU虚拟化)不同实现方式(纯软件、硬件辅助等)的基本原理，并以华为数据中心虚拟化解决方案(DCS)为例进一步介绍如何将这些技术应用到实践中。

3.2　CPU虚拟化

在现代计算机体系结构中，CPU(Central Processing Unit，中央处理器)是核心的运算和执行单元，负责解释和执行存储在内存中的指令，控制并协调其他硬件组件的运行。

CPU 的性能直接决定了计算机的整体运算能力，是数据处理和应用程序执行的关键。然而在传统的数据中心环境中，物理 CPU 资源往往得不到充分利用，造成资源的浪费。此外，物理 CPU 的管理和维护也相对复杂，难以满足日益增长的业务需求。为了解决这些问题，CPU 虚拟化技术应运而生。

CPU 虚拟化是计算虚拟化中的关键技术之一，它允许在单个物理 CPU（Physical CPU，pCPU）上创建多个虚拟 CPU（Virtual CPU，vCPU），每个 vCPU 都可以独立运行操作系统和应用程序。通过 CPU 虚拟化，可以将 pCPU 资源划分为多个逻辑单元，从而实现资源的灵活分配和高效利用。

本节首先概述 CPU 虚拟化的基本原理以及面临的一些挑战，接着从 CPU 所提供功能的角度介绍 CPU 虚拟化在主流体系架构（x86 和 ARM）中的实现方法，包括指令模拟、中断和异常的模拟及注入。

3.2.1 CPU 虚拟化基本原理

CPU 虚拟化的基本原理主要包括时分复用、指令集模拟、中断和异常处理等方面，其中时分复用原理是实现 CPU 共享的核心机制。

时分复用原理，简单来说，就是按照时间片轮转的方式，将 pCPU 的使用权分配给不同的 vCPU。在任意时刻，只有一个 vCPU 独占 CPU 资源以执行其任务，其他 vCPU 则处于等待状态。通过精确的时间片划分和调度算法，CPU 虚拟化技术能够确保每个 vCPU 都能够公平、高效地获得 CPU 资源，从而实现对 CPU 资源的共享和复用。

时分复用原理的实现依赖于虚拟化软件（Hypervisor）的支持。Hypervisor 作为虚拟机和物理硬件之间的中介层，负责管理和调度虚拟机的运行。当虚拟机需要执行指令时，它会向 Hypervisor 发出请求。Hypervisor 根据调度算法，将 CPU 资源分配给请求执行的 vCPU，并设置相应的时间片长度。在时间片内，vCPU 独占 CPU 资源执行其任务；时间片结束后，Hypervisor 会保存虚拟机的状态，并将其置于等待队列中，然后调度下一个虚拟机运行。

这种时分复用的方式有几个显著优势。首先，它提高了 CPU 资源的利用率。由于多个 vCPU 可以共享同一个 pCPU，因此可以充分利用 CPU 的计算能力，避免资源的浪费。其次，时分复用原理保证了 vCPU 的公平性和隔离性。通过合理的调度算法，可以确保每个 vCPU 都能够获得足够的 CPU 资源，而不会受到其他 vCPU 的影响。

以图 3-2 所示的一个简单例子来说明 CPU 虚拟化中的时分复用原理。假设系统有一个 pCPU 和三个 vCPU，每个 vCPU 都有一些任务需要执行。在初始化阶段，Hypervisor 将 pCPU 资源按照时间片的方式分配给这三个 vCPU。例如，可以设置每个虚拟机的时间片长度为 100ms。在时间片 1 内，CPU 资源分配给 vCPU1，它独占 CPU 执行其任务。当时间片 1 结束时，Hypervisor 保存 vCPU1 的状态，并将其置于等待队列中。然后，在时间片 2 内，CPU 资源分配给 vCPU2 执行其任务。同样地，当时间片 2 结束时，vCPU2 的状态被保存，并等待下一次调度。最后，在时间片 3 内，CPU 资源分配给 vCPU3 执行其任务。这个

过程不断循环进行，每个 vCPU 都有机会获得 CPU 资源并执行其任务。

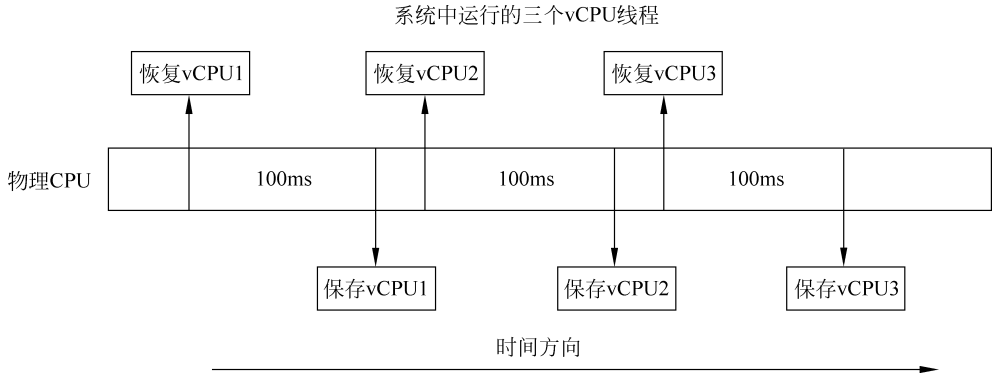


图 3-2　vCPU 时分复用 pCPU 示例

通过这种方式，CPU 虚拟化技术实现了物理 CPU 资源的共享和复用，提高了资源的利用率和灵活性。同时，精确的调度算法和隔离机制，确保了虚拟机在资源使用时的公平性和安全性。

上面简要描述了虚拟化环境中 vCPU 对 pCPU 的复用过程。vCPU 作为虚拟机内部的核心组件，其功能与物理机中的 pCPU 颇为相似。它负责读取指令，对指令进行译码并执行，同时能够处理来自虚拟存储器的数据，进行算术运算或逻辑运算，并将运算结果返回至虚拟存储器。除此之外，vCPU 还需承担虚拟机运行过程中异常情况和特殊请求的处理工作，及时响应系统事件，如磁盘数据读取完毕、网卡接收数据包等引发的中断。

综合来看，CPU 虚拟化面临两个核心挑战：一是如何高效且正确地执行虚拟机指令；二是如何及时响应各种系统事件。接下来将分别介绍针对这两个挑战的相关解决方案。

3.2.2　指令模拟

在 CPU 虚拟化中，指令的区分对于理解虚拟化机制至关重要。根据指令对系统资源和执行权限的要求，可以将指令分为特权指令、非特权指令和敏感指令三类。

（1）特权指令。只能在特定的权限级别下执行的指令，通常只有操作系统内核或具有特权的程序才能执行。这类指令通常用于执行对系统状态或资源的控制操作，例如，在 x86 架构中，HLT（暂停）指令和 CLI/STI（禁用/启用中断）指令都是特权指令。这些指令如果由非特权程序执行，可能会导致系统崩溃或数据损坏。因此，在虚拟化环境中，这些指令的执行通常会被 Hypervisor 捕获并模拟，以确保虚拟机的稳定运行。

（2）非特权指令。可以由普通应用程序执行的指令，这类指令主要用于数据处理、逻辑运算、内存访问等常规操作。由于这些指令不涉及对系统状态或资源的控制，因此它们可以在较低的权限级别下执行，而不会对系统的安全性和稳定性造成影响。例如，ADD、SUB、MUL 等算术运算指令，以及 MOV、CMP 等数据移动和比较指令都属于非特权指令。这些指令在虚拟机中可以直接执行，无须 Hypervisor 的介入，从而保证了应用程序的高效运行。

（3）敏感指令。虽然可以由普通应用程序执行，但执行时可能会触发特定安全或性能问题的指令。敏感指令则介于特权指令和非特权指令之间。如图 3-3 所示，所有的特权指令都是敏感指令，但不是所有的敏感指令都是特权指令。这些指令通常涉及对系统状态的间接修改或对特定资源的访问，如果不加以控制，可能会导致安全漏洞或性能下降。例如，在 Linux 系统中，sysenter 和 sysexit 指令用于在用户空间和内核空间之间进行快速切换，这些指令可以由用户空间程序执行，但它们的行为受到内核的严格控制。

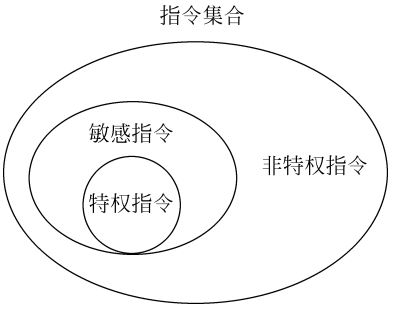


图 3-3　指令类型

指令模拟的核心思想是：在虚拟机执行指令时，如果该指令需要模拟（如特权指令），则由 Hypervisor 捕获该指令的执行，并将其转换为可在物理 CPU 上执行的指令序列。这一过程涉及指令的捕获、解码，模拟执行，以及结果返回。

（1）指令捕获。当虚拟机尝试执行一个需要模拟的指令时，CPU 的虚拟化支持机制会触发一个异常或中断，将控制权交给 Hypervisor。这一过程通常是通过设置特定的 CPU 标志位或使用专门的虚拟化指令来实现的。

（2）指令解码。Hypervisor 捕获到需要模拟的指令后，首先会对该指令进行解码，分析其操作码、操作数等信息。这一步骤对于理解指令的语义和执行方式是至关重要的。

（3）模拟执行。根据解码得到的指令信息，Hypervisor 会模拟执行该指令。对于特权指令，Hypervisor 会执行相应的操作以改变虚拟机的系统状态或访问敏感资源；对于非特权指令，Hypervisor 则可以直接在物理 CPU 上执行该指令，并将结果返回给虚拟机。在模拟执行过程中，Hypervisor 还需要处理一些特殊情况，如虚拟机的中断和异常处理、内存访问的权限检查等。这些都需要 Hypervisor 与虚拟机之间进行紧密的协作和同步。

（4）结果返回。模拟执行完成后，Hypervisor 将执行结果返回给虚拟机。如果模拟的是特权指令，则 Hypervisor 还需要更新虚拟机的系统状态或敏感资源的状态，以确保虚拟机对系统状态的感知与物理机一致。

指令模拟的实现可以大致划分为软件和硬件辅助两大类解决方案，具体内容如下所述。

1. 指令模拟的软件解决方案

软件解决方案主要依赖于虚拟化软件（如 Hypervisor）来实现指令模拟，通过软件层面的模拟执行，能够实现对各种指令的解码、模拟执行和结果返回。其中最常见的技术包括陷入模拟（Trap and Emulate）和二进制翻译（Binary Translation）。

1）陷入模拟

陷入模拟是一种简单的指令模拟方法，其流程如图 3-4 所示。对于非敏感指令，pCPU 直接解码并处理其请求，相关效果随即反映至物理寄存器上。当虚拟机尝试执行一条敏感指令（如特权指令或 I/O 指令）时，该指令会触发一个异常或陷阱，导致虚拟机陷入 Hypervisor

中。Hypervisor随后会模拟执行这条指令，并将结果返回给虚拟机。从程序的角度来看就是一组数据结构与相关处理代码的集合。数据结构用于存储虚拟寄存器的内容，而相关处理代码负责按照pCPU的行为将效果反映到虚拟寄存器上。因此，陷入模拟的目标是让虚拟机里执行的敏感指令陷入下来后能被Hypervisor模拟，而不要直接作用于真实硬件上。

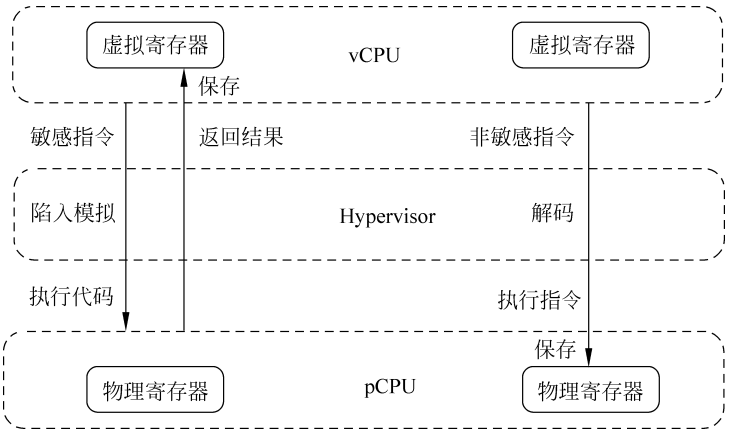


图3-4　虚拟化环境下指令陷入模拟示意图

一个实际产品的例子是KVM。KVM是Linux平台上的一个开源虚拟化系统，它利用Linux内核的虚拟化功能来实现高效的CPU虚拟化。在KVM中，当虚拟机执行敏感指令时，会触发一个陷阱(trap)，将控制权转交给用户空间的QEMU进程。QEMU随后会模拟执行这条指令，并将结果返回给KVM和虚拟机。通过陷入模拟的方式，KVM能够在保证虚拟机安全性的同时，提供高效的虚拟化性能。

2）二进制翻译

二进制翻译是另一种指令模拟技术。它通过对虚拟机的指令流进行动态翻译和优化，将其转换为宿主机的指令集，从而在宿主机上执行。二进制翻译技术能够减少陷入模拟带来的性能开销，提高虚拟机的执行效率。

一个典型的二进制翻译技术的实现是QEMU。QEMU是一个跨平台的开源机器模拟器和虚拟化器，它支持多种处理器架构和操作系统。QEMU使用动态二进制翻译技术，将虚拟机的指令流转换为宿主机的指令集，并在宿主机上执行。QEMU还采用了即时编译(JIT)技术，对频繁执行的代码段进行优化和缓存，进一步提高执行效率。

2. 指令模拟的硬件辅助解决方案

尽管软件解决方案在指令模拟方面取得了一定的成果，但其性能开销仍然是一个不可忽视的问题，因为软件解决方案需要实时解码、模拟和执行虚拟机的指令，这增加了额外的计算负担。为了进一步提高虚拟化的性能，硬件厂商推出了许多硬件辅助虚拟化技术，其中最具代表性的是Intel的VT-x(Virtualization Technology for x86)、AMD的AMD-V技术和ARMv8架构中的虚拟化扩展。

1）x86 架构的硬件辅助虚拟化解决方案

Intel 在原有 x86 CPU 基础上增加了 VMX（Virtual Machine eXtension，虚拟机扩展）模式来实现 CPU 的硬件虚拟化，CPU 可以通过 VMXON/VMXOFF 指令打开或关闭 VMX 操作模式。VMX 模式中定义了 VMM（Virtual Machine Monitor）以及 VM（Virtual Machine）；为适配 x86 芯片虚拟化，芯片新增了根模式（root mode）和非根模式（non-root mode）。两个模式中都具备 ring0～ring3 运行级别，VMX 模式的整体系统架构与对应的运行级别如图 3-5 所示。

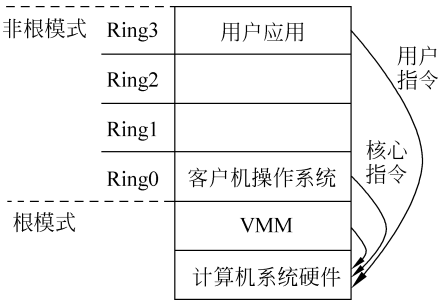


图 3-5 VMX 模式的整体系统架构与对应的运行级别

在根模式下，VMM 可以管理虚拟机的创建、销毁和调度等操作；而在非根模式下，虚拟机则执行其正常的指令集。这种模式的切换由硬件自动完成，无须软件中断或陷入模拟，从而大大提升了性能。在 VT-x 的支持下，VMM 可以利用专门的指令集来捕获和控制虚拟机的指令执行。当虚拟机执行一个需要模拟的指令时，VT-x 可以将其直接陷入 VMM 中，而无须像传统陷入模拟那样通过软件中断来实现。这大大减少了陷入开销，提高了虚拟机的性能。

2）ARM 架构的硬件辅助虚拟化解决方案

在 ARM 架构的硬件辅助虚拟化技术中，ARMv8 架构特别引入了专门的虚拟化拓展——VHE（Virtualization Host Extension），这显著增强了 ARM 处理器对虚拟机指令模拟的高效支持。VHE 作为 ARM 架构专为提升虚拟化性能而设计的硬件特性，其核心思想在于精简虚拟化流程中的冗余上下文切换与陷阱操作，进而优化虚拟机的运行效率。

具体来说，VHE 通过引入新的处理器模式——EL2（Exception Level 2，异常级别 2），使得 Hypervisor 能够直接运行在这一模式下，对虚拟机进行高效管理（如图 3-6 所示）。当虚拟机发出指令或遭遇异常时，这些操作会被直接捕获到 EL2，而无须像传统虚拟化那样先陷阱到 EL1（异常级别 1）再进行处理。这种直接捕获机制大大减少了虚拟化过程中的开销，提升了整体性能。

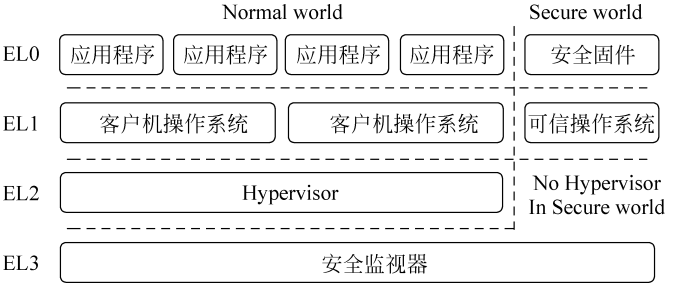


图 3-6 ARM 异常级别分布

此外，VHE 还提供了丰富的寄存器和指令集，以支持虚拟机的创建、配置和销毁等操作。这些硬件特性的加入使得 CPU 虚拟化过程更为流畅、高效。

总的来说，VHE 原理的核心在于通过硬件级别的优化，减少虚拟化过程中的开销，提升虚拟机的运行效率。这种优化不仅体现在 CPU 指令的执行上，还体现在虚拟机与 Hypervisor 之间的交互过程中。

综合来看，软件解决方案和硬件辅助解决方案在指令模拟中各有优势。软件解决方案的灵活性使其能够适应各种虚拟化场景，而硬件辅助解决方案的高效性则能够显著提升虚拟机的性能。在实际应用中，可以根据具体需求和场景选择适合的解决方案，或者结合两者使用，以达到最佳的虚拟化效果。

3.2.3　中断虚拟化

在计算机系统中，中断与异常均扮演着至关重要的角色，它们作为处理外部事件或内部异常状况的核心机制而存在。中断，通常源于外部设备或信号的触发，旨在通知 CPU 暂停当前执行的程序，转而执行中断服务程序；异常，则是由 CPU 内部产生，往往源于程序执行过程中遭遇的错误或特殊状况，如除零错误、地址越界等，亦可视为一种特殊的中断事件。中断与异常共同构成了 CPU 与外部世界或内部环境交互的桥梁，确保系统能够稳定运行并高效响应外部事件，是计算机系统中不可或缺的关键机制。

一个常见的中断例子是键盘输入中断。当用户按下键盘上的某个键时，键盘控制器会检测到这个动作，并产生一个中断信号发送给 CPU。CPU 在接收到这个中断信号后，会暂停当前正在执行的程序，保存现场信息（如程序计数器、寄存器状态等），然后跳转到中断服务程序进行处理。中断服务程序会读取键盘控制器的输入缓冲区，获取用户输入的键值，并根据需要更新内存中的相关数据结构。最后，中断服务程序会恢复现场信息，并返回原程序继续执行。通过这种方式，系统能够实时响应用户的键盘输入，实现人机交互。

在虚拟化环境中，中断虚拟化是实现 CPU 虚拟化的重要环节。由于虚拟机与宿主机共享物理 CPU 资源，因此需要一种机制来确保虚拟机能够正确地接收和处理中断与异常，同时保持虚拟机的隔离性和安全性。

中断虚拟化的核心原理在于 Hypervisor 对中断信号的捕获、分类、模拟和注入。以下是中断虚拟化过程的主要步骤。

（1）中断捕获。当物理处理器接收到中断信号时，虚拟化层首先会捕获这个中断。捕获中断后，虚拟化层会根据中断的类型和来源，判断该中断应该被路由到哪个虚拟机。

（2）中断分类。虚拟化层会对捕获到的中断进行分类。根据中断的来源，可以将中断分为外部设备中断（如硬盘、网卡等）和软件中断（如定时器中断、异常处理等）。根据中断的目的地，可以将中断分为针对特定虚拟机的中断和针对所有虚拟机的全局中断。

（3）中断模拟。对于需要传递给虚拟机的中断，虚拟化层会模拟中断信号，使其看起来像是直接来自物理处理器。中断信号的模拟包括设置虚拟机的中断向量表、保存中断发生时的处理器状态等。

（4）中断注入。虚拟化层将模拟好的中断信号注入目标虚拟机的非根模式中。这样，虚拟机就能够像处理物理中断一样处理这个虚拟中断。中断注入的过程需要确保虚拟机不会意识到它是在虚拟化环境中运行的。

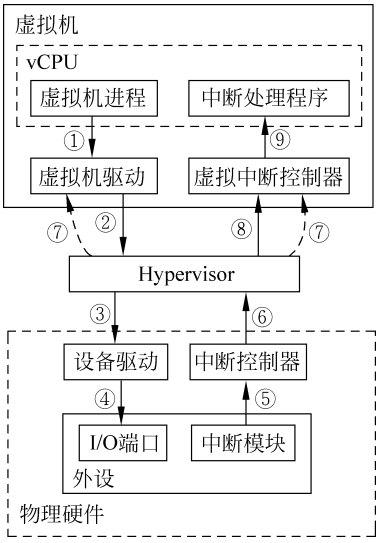


图 3-7　虚拟中断注入的完整流程

接下来以虚拟机 I/O 过程的虚拟中断注入为例，详细展示虚拟中断注入的完整流程。如图 3-7 所示，虚拟机的虚拟中断注入通常包含如下步骤。

（1）虚拟机内的应用程序通过系统调用方式，向虚拟机驱动程序发起 I/O 请求（如图 3-7 中标号①所示）。

（2）虚拟机驱动程序在访问虚拟设备（例如虚拟硬盘）时，触发虚拟机陷入 Hypervisor 进行处理（如图 3-7 中标号②所示）。

（3）Hypervisor 调用宿主机设备驱动（如图 3-7 中标号③所示），并通过读写物理外设（如物理硬盘）提供的 I/O 端口，向物理外设发起 I/O 操作（如图 3-7 中标号④所示）。

（4）当 I/O 操作完成后，物理外设的中断模块会向物理中断控制器发送一个中断信号（如图 3-7 中标号⑤所示）。

（5）Hypervisor 执行物理驱动程序所注册的中断服务程序（Interrupt Service Routine，ISR）（如图 3-7 中标号⑥所示）。

（6）设置虚拟设备和虚拟中断控制器相应寄存器的状态（如图 3-7 中标号⑦所示）。

（7）Hypervisor 通过上下文切换机制恢复虚拟机的运行（如图 3-7 中标号⑧所示）。

（8）在虚拟机恢复运行时，其 vCPU 会检查虚拟中断控制器中是否存在待处理的中断，并调用相应的中断服务程序进行处理（如图 3-7 中标号⑨所示）。

综上所述，在虚拟化环境中，实现外部中断注入通常是通过以下步骤完成的：将中断信号传递给 Hypervisor 进行处理；Hypervisor 负责配置虚拟中断控制器，并将模拟的虚拟中断信号注入相应的虚拟机中。这一机制确保了虚拟机在虚拟化环境中能够准确无误地响应并处理外部中断，从而维持其正常运行状态。

总体而言，中断虚拟化是一个复杂的过程，涉及中断和异常的定义、中断源与 Hypervisor 之间的交互机制以及中断的模拟与注入流程。值得注意的是，不同的 Hypervisor 在中断虚拟化方面的设计和实现方式可能有所差异，同时，不同的物理设备以及虚拟机操作系统也会对中断虚拟化的实现产生一定影响。因此，在探讨中断虚拟化时，需要综合考虑各种因素。

下面将分别介绍在 x86 和 ARM 这两种不同的计算机体系架构下，中断虚拟化的具体实现机制。虽然这些机制的实现方式各异，但都旨在确保虚拟机在虚拟化环境中能够高效地处理中断，从而提供稳定可靠的虚拟化服务。

1）x86 架构 Intel VT-x 中断虚拟化

在 x86 架构下，虚拟机中断架构如图 3-8 所示，其中包含了虚拟 PIC、虚拟 I/O APIC 以及虚拟 Local APIC 三个核心组件。以下是对这三个关键组件含义和作用的阐释。

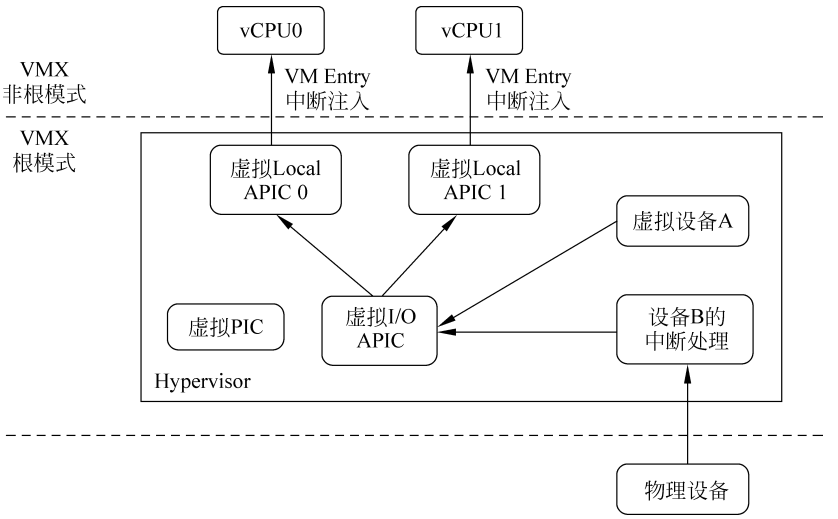


图 3-8　x86 架构下虚拟机中断架构

（1）虚拟 PIC：PIC（Programmable Interrupt Controller，可编程中断控制器）是早期计算机系统中用于处理中断的控制器。在虚拟化环境中，虚拟 PIC 模拟了物理 PIC 的行为，使得虚拟机能够接收到并处理来自外部设备的中断信号。

（2）虚拟 I/O APIC：I/O APIC（Advanced Programmable Interrupt Controller，高级可编程中断控制器）是更先进的中断控制器，用于处理来自 I/O 设备的中断。在虚拟化环境中，虚拟 I/O APIC 负责将中断路由到目标虚拟机。通过模拟 I/O APIC 的行为，虚拟 I/O APIC 确保了中断能够正确、高效地传递给虚拟机。

（3）虚拟 Local APIC：Local APIC 是每个处理器核心上的本地中断控制器，负责处理本地中断和从 I/O APIC 接收的中断。在虚拟化环境中，每个虚拟机都有一个与之关联的虚拟 Local APIC。虚拟 Local APIC 模拟了物理 Local APIC 的行为，使得虚拟机能够像物理机一样处理中断。

与虚拟 CPU 类似，虚拟 PIC、虚拟 I/O APIC 以及虚拟 Local APIC 均是由虚拟机监控器 Hypervisor 维护的软件实体。每个 vCPU 均对应一个虚拟 Local APIC，用于接收中断；同时也包含虚拟 I/O APIC 或虚拟 PIC，用于发送中断。当虚拟设备需要发送中断时，它会调用虚拟 I/O APIC 的接口来发送中断请求。虚拟 I/O APIC 会根据中断请求，挑选出相应的虚拟 Local APIC，并调用其接口发送中断请求。随后，虚拟 Local APIC 会进一步利用 VT-x 的事件注入机制，将中断注入相应的 vCPU 中，从而完成整个中断处理流程。

x86 架构下的中断虚拟化，其核心任务在于实现一套完备的虚拟中断架构，这包括虚拟 PIC、虚拟 I/O APIC 及虚拟 Local APIC 等关键组件，并需确保中断的生成、采集及注入过

程得以顺利执行。

（1）中断采集。

中断采集即将虚拟机内的设备中断请求有效传递至虚拟中断控制器。在虚拟化环境下，客户机的中断主要源自两方面：①软件模拟的虚拟设备（如模拟串口），此类设备在特定条件下能够生成虚拟中断；②直接分配给客户机的物理设备（如物理网卡），其产生的物理中断需经过特殊处理。

对于虚拟设备而言，其作为软件模块，在需要发出中断请求时，可通过调用虚拟中断控制器提供的接口函数实现，如利用虚拟 PIC 或虚拟 I/O APIC 的接口。

对于直接分配的物理设备，情况则变得较为复杂。当物理设备发生中断时，其处理程序通常位于虚拟机操作系统内部。然而，在虚拟化环境中，物理中断控制器由 Hypervisor 所控制。因此，物理中断首先需由 Hypervisor 的中断处理程序接收，再经由特定机制注入虚拟机。

（2）中断注入。

中断注入将虚拟中断控制器所采集的中断请求，依据其优先级顺序，逐一注入客户机的虚拟处理器。此过程涉及两个关键问题：①如何确定并获取需注入的最高优先级中断的相关信息；②如何实现将中断有效地注入虚拟机 vCPU。

对于第一个问题，虚拟中断控制器负责将中断按优先级排序，Hypervisor 通过调用其提供的接口函数，即可获取当前最高优先级的中断信息。

对于第二个问题，虚拟 Local APIC 提供了将中断注入虚拟机 vCPU 的基础功能。Hypervisor 通过调用虚拟 Local APIC 的接口实现中断注入。

上述机制可确保 x86 架构下的中断在虚拟化环境中被高效、准确地处理。

2）ARM 架构 GIC 中断虚拟化

GIC（Generic Interrupt Controller，通用中断控制器）作为 ARM 公司所提供的统一中断控制器架构，阐释了 ARM 平台上中断控制器内部分发逻辑（Distributor）与 CPU 接口（CPU Interface）的标准化设计。在 GIC 的演进过程中，GICv3 引入了 ITS（Interrupt Translation Service，终端转换服务）的支持，而 GICv4 则进一步引入了 LPIs（Locality-specific Peripheral Interrupts，本地特定外设中断）中断透传功能，显著提升了中断处理的灵活性与效率。

自 GICv2 起，该架构开始支持中断虚拟化的硬件扩展。vGIC 为每个 CPU 引入了一个专用的 vGIC CPU 接口及相应的 Hypervisor 控制接口，从而允许虚拟机直接利用 vGIC CPU 接口进行中断处理。当非安全物理中断发送给特定 CPU 时，该 CPU 将被触发进入 EL2 模式。此时，Hypervisor 上的中断处理程序将依据物理中断对应的 VMID 等信息，通过已注册的中断列表进行配置，并将虚拟中断发送至 vGIC CPU 接口，进而路由至相应的 vCPU。与 x86 架构类似，以下将详细阐述 ARM 架构下的中断采集与中断注入两大关键流程。

（1）中断采集。

在 ARM 架构中，虚拟中断的来源主要有两个：①通过配置特定寄存器（如图 3-9 中的 HCR_EL2 寄存器），内部 CPU 核能够产生中断信号，此时 Hypervisor 将模拟中断控制器向 vCPU 发送中断信号；②利用 GICv2 及其后续版本的外部中断控制器来生成虚拟中断，进一步丰富中断信号的来源与形式。

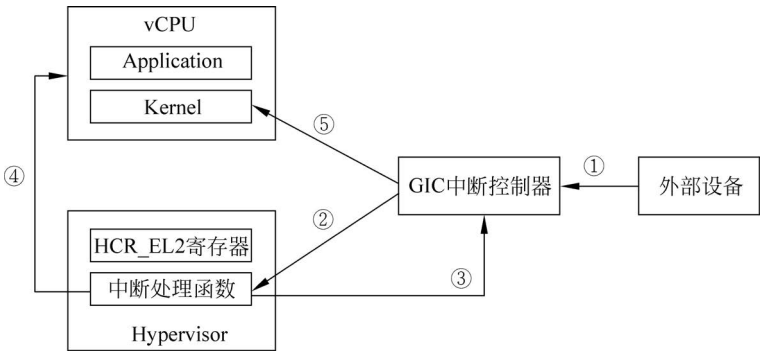


图 3-9　ARM 架构中断注入流程

（2）中断注入。

ARM 架构下中断注入 vCPU 的流程如下：①外部设备向 GIC 中断控制器发送中断信号；②GIC 中断控制器响应并产生物理中断异常。由于配置了 HCR_EL2 寄存器，这些中断被定向至 EL2 层。Hypervisor 识别出触发中断的外部设备，并确定其已被分配给某个虚拟机。随后，Hypervisor 通过中断处理函数判断应将中断转发至哪个 vCPU。

① Hypervisor 配置 GIC，使其以虚拟中断的形式将物理中断转发至目标 vCPU。

② Hypervisor 将控制权交还给 vCPU。

③ vCPU 在 EL0 或 EL1 层级运行，并接收来自 GIC 的虚拟中断，进而执行相应的中断处理逻辑。

在纯虚拟中断的情境下，Hypervisor 可直接将中断注入 vCPU，无须经过外部设备或 GIC 中断控制器的中介。

本节已经对 CPU 虚拟化的基本原理、关键技术和实际应用进行了深入探讨。CPU 虚拟化不仅提升了服务器的资源利用率，还显著增强了系统的可靠性和灵活性，为企业和个人用户带来了前所未有的便利。然而，虚拟化技术的魅力远不止于此，接下来将对另一个重要领域——内存虚拟化进行解析。

3.3　内存虚拟化

在计算虚拟化技术中，内存虚拟化是至关重要的一环。它允许虚拟机共享同一台物理机上的物理内存资源，同时保持每个虚拟机内存空间的独立性和隔离性。

在深入探讨内存虚拟化技术之前，有必要先对计算机系统中的进程内存管理机制进行

介绍。从进程内存管理的视角来看，由于计算机系统的物理地址空间具有唯一性，为确保多个进程能够高效且彼此隔离地使用这些资源，操作系统引入了虚拟地址空间（Virtual Address，VA）的概念。此虚拟地址空间通过映射机制与物理地址空间（Physical Address，PA）的特定部分或整体相对应。如图 3-10 所示，操作系统为不同进程分配了不同的物理内存块，而它们均使用相同的虚拟地址进行访问，共享使用物理内存的同时又避免了访问冲突。

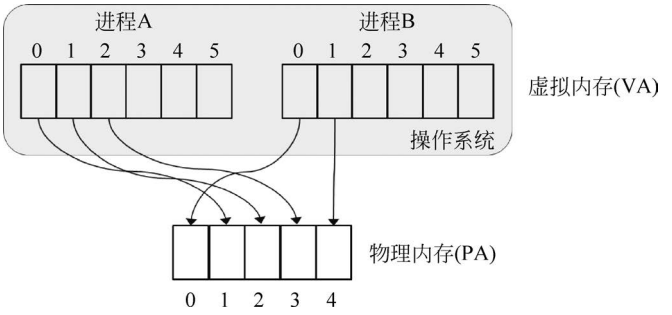


图 3-10　虚拟内存与物理内存映射示意图

一个自然的问题是：虚拟内存的本质是什么呢？虚拟内存的核心思想是将内存地址空间划分为用户空间和内核空间。用户空间是应用程序直接访问的内存区域，而内核空间则是操作系统内核使用的内存区域。这种划分保证了系统的稳定性和安全性，因为操作系统内核可以限制应用程序对物理内存的访问权限。

页表（Page Table，PT）和内存管理单元（Memory Manage Unit，MMU）是与虚拟内存实现紧密相关的两个重要概念。

（1）页表。页表是一种数据结构，其核心功能在于将虚拟内存地址映射至物理内存地址。在使用虚拟内存的系统中，每个进程均拥有独立的页表，用以管理其虚拟地址空间与物理内存之间的映射关系。页表详细记录了虚拟页面与物理页面之间的对应关系，从而使操作系统能够将虚拟地址高效地转换为物理地址。

（2）内存管理单元。MMU 是一种硬件组件，负责在程序运行时执行虚拟地址到物理地址的转换任务。它是计算机体系结构中的核心组成部分，通常内嵌于 CPU 中。MMU 通过读取页表的内容，依据虚拟地址找到对应的物理地址，并将其传递至系统总线，从而实现高效的虚拟内存管理。

在虚拟地址到物理地址的转换过程中，MMU 扮演着至关重要的角色。如图 3-11 所示，它通过访问当前进程的页表，根据虚拟地址查找对应的物理地址，并将该物理地址传递至内存系统，以便获取相应的数据或指令。因此，页表成为 MMU 执行地址转换所依赖的关键数据结构。

然而，在虚拟化环境中，传统的进程内存管理机制遭遇前所未有的挑战。虚拟机需共享有限的物理内存资源，同时确保各自内存空间的独立性。因此，我们迫切需要一种更为先进的内存管理机制——内存虚拟化，以实现对虚拟机内存访问的有效管理。

下面将首先深入剖析内存虚拟化的基本原理及其所面临的核心问题，随后介绍在主流

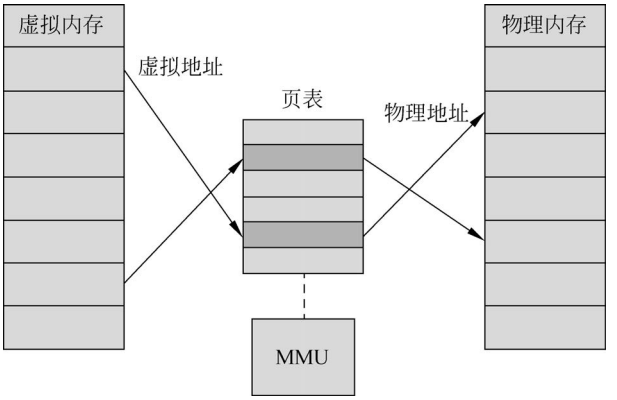


图 3-11　分页机制下虚拟地址与物理地址转换

计算机系统架构(x86 和 ARM)中基于硬件辅助的内存虚拟化的实现方式,以期为读者提供更为全面和深入的展示。

3.3.1　内存虚拟化基本原理

内存虚拟化的核心理念,在于为每台虚拟机构建一个独立的虚拟地址空间,并通过 Hypervisor 实现虚拟地址到物理地址的转换。因此,虚拟机在运行过程中,虽然看似独占了全部物理内存,实则通过 Hypervisor 的映射机制,与其他虚拟机共享内存资源。

在物理环境中,操作系统直接管理物理内存,而应用程序则通过操作系统提供的接口进行物理内存的访问。如图 3-12 所示,此时内存地址的应用层次相对简单,主要涵盖应用程序虚拟地址(Virtual Address,VA)和物理地址(Physical Address,PA)。

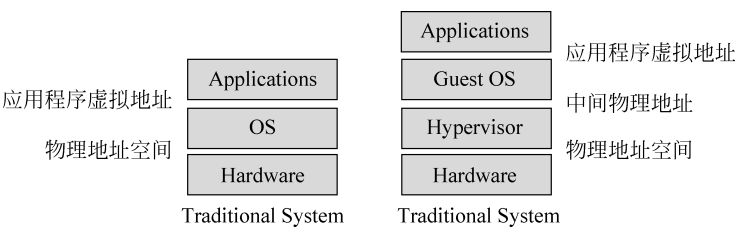


图 3-12　物理环境和虚拟化环境下内存地址应用层次

然而,在虚拟化环境中,内存地址的应用层次则显得更为复杂。除了应用程序虚拟地址和物理地址之外,还引入了客户机虚拟地址(Guest Virtual Address,GVA)和客户机物理地址(Guest Physical Address,GPA)的概念。如图 3-13 所示,虚拟机内的应用程序利用客户机的虚拟地址进行内存访问,这些虚拟地址在虚拟机内部首先转换为客户机物理地址。随后,虚拟化软件再将客户机物理地址进一步转换为宿主机的物理地址,最终达成对物理内存的访问。这种多层次的地址转换与映射机制,不仅确保了虚拟机之间内存的隔离,还大大提升了系统的安全性。

由于内存虚拟化引入了客户机物理地址这一虚拟的地址层次,虚拟机在运行时所感知

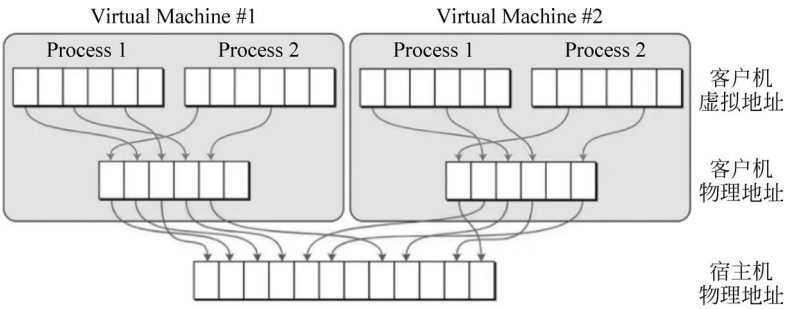


图 3-13 “虚拟”物理内存示意图

和使用的内存地址，实际上并非直接对应宿主机的物理内存地址。这种地址空间的虚拟化带来了诸多便利，如资源隔离、动态分配和灵活管理，但同时也带来了一系列需要解决的复杂问题。在实现虚拟机对实际物理内存使用的过程中，内存虚拟化主要需要处理以下两大核心问题。

(1) 客户机物理地址与宿主机物理地址间的映射问题。

客户机物理地址，即为虚拟机操作系统所观察到的内存地址，而宿主机物理地址则指代实际物理内存中的地址。鉴于虚拟机的内存空间要经虚拟化处理，因此客户机物理地址与宿主机物理地址之间并无直接的映射关系。为使虚拟机能够正确地访问并有效地利用内存资源，必须构建一套映射机制，以实现客户机物理地址至宿主机物理地址的转换。

在内存虚拟化的框架内，虚拟内存技术得以进一步拓展，以支持多个虚拟机共享同一物理内存资源。每个虚拟机均拥有独立的虚拟地址空间，并通过其专属的 MMU 实现地址转换。虚拟化软件(如 Hypervisor)负责对这些虚拟机的内存资源进行统一管理，确保它们互不干扰，并尽可能高效地利用物理内存。

虚拟化软件维护着一组或多组虚拟页表，这些页表精确地反映了虚拟机对内存的使用状况。当虚拟机发起内存访问请求时，虚拟化软件将拦截这些请求，并根据虚拟页表进行相应的地址转换。若虚拟机请求的页面未驻留于物理内存中，虚拟化软件或将触发页面置换(page swapping)操作，从其他虚拟机或交换空间中获取所需页面。

维护这种映射关系的过程颇为复杂。虚拟机在运行过程中会频繁进行内存访问，页表亦需不断更新与同步，以确保映射关系的精确性与实时性。通过这种机制，内存虚拟化不仅实现了内存的隔离性(每个虚拟机拥有独立的内存视图)，还提高了内存的利用率(通过共享和动态分配物理内存资源)。

(2) 客户机物理地址访问的截获问题。

在虚拟机环境中，操作系统发出的内存访问请求实际上是发往 Hypervisor 的。为确保虚拟机正确地访问物理内存，Hypervisor 必须截获这些请求，并根据事先建立的映射关系进行地址转换。

截获客户机物理地址访问的过程，通常涉及陷阱或中断(interrupt)机制。当虚拟机尝

试访问某个内存地址时，若该地址不在当前映射范围内或访问权限不符，便会触发陷阱或中断。此时，Hypervisor 将介入处理，截获该访问请求，并依据页表或其他映射机制进行地址转换。

此截获与转换过程需要高效且精确。若处理速度迟缓或转换有误，将导致虚拟机性能下降或内存访问失败。因此，Hypervisor 需采用高效的算法和数据结构以维护映射关系，并优化截获与转换过程的性能。

同时，为确保截获的透明性与兼容性，Hypervisor 也需要处理一些特殊情况。例如，当虚拟机使用特定的内存访问指令或操作时，Hypervisor 需能够识别并正确处理，以保障虚拟机的正常运行。

本节介绍了内存虚拟化如何解决客户机物理地址与宿主机物理地址间的映射以及对客户机物理地址访问的截获问题。在 3.3.2 节中，将结合 x86 和 ARM 架构的具体技术，阐述如何实现内存虚拟化。

3.3.2　硬件辅助的内存虚拟化

1）x86 扩展页表

Intel EPT（扩展页表）与 AMD NPT（嵌套页表）作为硬件辅助内存虚拟化的典型代表，均针对内存管理单元（MMU）进行了虚拟化扩展。两者在原理上相似，本节着重探讨 EPT 技术。

EPT 技术的实现离不开特定的硬件支持。首先，Intel 处理器提供了与 EPT 相关的指令集和寄存器，这些指令集和寄存器用于管理 EPT。其次，Hypervisor 承担起维护 EPT 页表的重任，它负责建立虚拟机虚拟地址与宿主机物理地址之间的映射关系。当虚拟机发起内存访问请求时，处理器会根据 EPT 进行地址转换，确保虚拟机能够准确无误地访问物理内存。

如图 3-14 所示，EPT 的基本原理在于在原有 CR3 页表地址映射（整个页表的基址存放在 CR3 寄存器里。CR3 存放的是物理地址，这是整个地址转换最根本的基础）的基础上引入 EPT，实现另一层映射。EPT 的引入，实现了从虚拟机虚拟地址（GVA）到虚拟机物理地址（GPA），再到宿主机物理地址（HPA）的两次地址转换，且均由硬件完成，从而显著提升了内存虚拟化的性能。

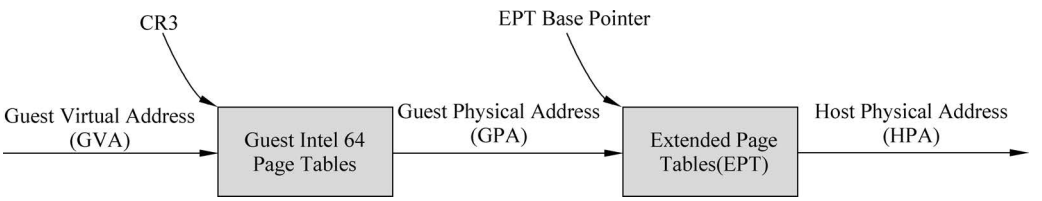


图 3-14　EPT 基本原理图

在实现过程中，EPT 技术还充分考虑了安全性和性能的优化。例如，EPT 支持页级别

的权限控制，有效防止了虚拟机之间的非法内存访问。同时，EPT 采用多级页表结构，不仅减小了页表的大小和内存占用，还提高了内存访问的效率。

综上所述，EPT 技术为内存虚拟化提供了强大的硬件支持。通过硬件级别的地址转换和权限控制，EPT 技术确保了虚拟机在共享物理内存时的安全性和性能。

2）ARM 两阶段页表

ARM 架构在内存虚拟化方面同样提供了对硬件辅助内存虚拟化的支持，下面将以目前主流的 ARM v8 为例，简要介绍内存虚拟化在 ARM 架构中的实现。

ARM v8 架构下的内存虚拟化采用了两阶段页表转换（two-stage page table translation）机制，这种机制增强了安全性和灵活性，同时优化了性能。关于这一机制的工作原理详细阐述如下。

首先，需要理解内存虚拟化在 ARM v8 架构中的基本目标：允许虚拟机（Guest OS）在其自身的地址空间中运行，同时确保这些地址空间被正确映射到物理内存上，且不同虚拟机之间的内存空间是相互隔离的。为了实现这一目标，ARM v8 引入了两阶段页表转换机制。

如图 3-15 所示，两阶段页表转换的第一阶段（Stage 1）在虚拟机内部进行。虚拟机维护自己的页表，这些页表将虚拟机的虚拟地址转换为中间物理地址（Intermediate Physical Address，IPA）。这个转换过程发生在虚拟机内部，由虚拟机的页表管理单元（Page Table Management Unit，PTMU）负责。

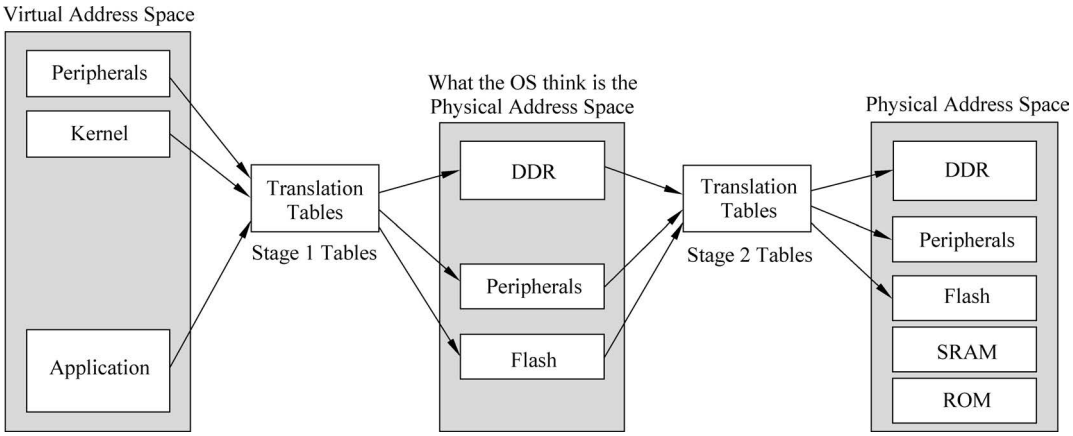


图 3-15　ARM v8 两阶段页表的地址转换

第二阶段（Stage 2）转换发生在宿主机（Host OS）层面。宿主机维护一个转换表，通常称为第二级页表或宿主机页表，它将中间物理地址转换为最终的物理地址（Physical Address，PA）。这一阶段的转换由宿主机的内存管理单元（MMU）执行。

这种两阶段转换机制带来了几个重要的优势。

（1）安全性。由于虚拟机仅能访问其自身页表，无法直接触及宿主机页表或物理内存，

因此实现了内存空间的隔离。这有效防止了虚拟机间的潜在干扰与攻击。

(2) 灵活性。通过调整宿主机页表,宿主机能够轻松变更虚拟机的内存映射,从而实现诸如内存热插拔、内存气球等高级功能。

(3) 性能优化。尽管两阶段转换相较于单一阶段转换增加了复杂性,但 ARM v8 通过优化算法与硬件支持,确保了性能损失的最小化。

综上所述,ARM v8 架构下的内存虚拟化两阶段页表转换机制,通过虚拟机与宿主机页表转换的协同工作,实现了安全、灵活且高效的内存虚拟化。

除了两阶段页表转换技术之外,ARM 架构还借助其他硬件特性来优化内存虚拟化的性能。例如,ARM 处理器支持快速上下文切换技术,大幅地减少了虚拟机间切换时的开销;同时,ARM 架构提供对内存压缩和加密等高级功能的支持,进一步提升了虚拟化环境的整体性能与安全性。

3.4　I/O 虚拟化

I/O 虚拟化是计算虚拟化技术的重要组成部分,它主要解决的是虚拟机与外部 I/O 设备之间的交互问题,是计算机与外部世界进行信息交换的桥梁,涵盖从键盘、鼠标的输入到显示器、打印机的输出,再到网络通信、磁盘存储等复杂的数据交换过程。

在传统的物理机环境中,每个设备直接与操作系统进行交互,操作系统负责管理和调度这些设备的 I/O 操作。然而,在虚拟化环境中,情况变得复杂起来。多个虚拟机共享同一套物理硬件资源,如何确保每个虚拟机都能高效、安全地访问外部设备,成为虚拟化技术需要解决的关键问题。

I/O 虚拟化技术应运而生,它通过在虚拟机与物理设备之间引入虚拟化的 I/O 层,实现对 I/O 操作的抽象和隔离。虚拟化的 I/O 层负责接收虚拟机的 I/O 请求,并将其转换为物理设备可以理解的指令,从而实现了虚拟机对物理设备的透明访问。

I/O 虚拟化不仅提高了虚拟机的 I/O 性能,还增强了系统的安全性和灵活性。通过精细的权限控制和资源调度,I/O 虚拟化可以确保每个虚拟机只能访问其授权的设备资源,防止了非法访问和数据泄露的风险。同时,I/O 虚拟化还支持动态资源分配和故障隔离,提高了虚拟化环境的可用性和稳定性。

相比于 CPU 和内存,I/O 设备的类型更加多样化,因此 I/O 虚拟化的方式也层出不穷。本节首先介绍 I/O 虚拟化的基本原理,然后对 I/O 虚拟化的具体实现方式进行阐述,包括基于软件实现的 I/O 虚拟化、I/O 半虚拟化和硬件辅助的 I/O 虚拟化。

3.4.1　I/O 虚拟化基本原理

为了便于理解 I/O 虚拟化原理,下面先给出 I/O 接口的基本概念。I/O 接口主要涉及处理器、内存和 I/O 设备之间的数据交互。根据数据传输的方式,I/O 接口可以分为端口

I/O(Port I/O,PIO)、内存映射 I/O(Memory Map I/O,MMIO)和直接内存访问(Direct Memory Access,DMA)三种。

1) 端口 I/O(PIO)

PIO 是最早的 I/O 接口,它通过处理器执行专门的 I/O 指令来完成数据的传输。在 PIO 方式中,处理器首先向 I/O 设备发送一个 I/O 指令,告诉设备要读取或写入的数据地址和数据长度。然后,处理器等待设备准备好数据后,再通过多次的读/写操作来完成数据的传输。PIO 方式的缺点是效率较低,因为每次数据传输都需要处理器的参与。

2) 内存映射 I/O(MMIO)

MMIO 是一种将 I/O 设备的地址空间映射到处理器的内存地址空间的 I/O 接口。在 MMIO 方式中,处理器可以直接通过内存访问指令来访问 I/O 设备的地址空间,从而实现数据的传输。MMIO 方式提高了 I/O 操作的效率,因为处理器可以直接访问内存,而不需要执行专门的 I/O 指令。但是,MMIO 方式需要处理器和 I/O 设备之间的地址空间进行映射,这增加了系统的复杂性。

3) 直接内存访问(DMA)

DMA 技术允许 I/O 设备直接与内存进行数据传输,而无须处理器的直接参与,从而极大地提升了数据传输的效率。现以磁盘 I/O 为例,描述 DMA 操作的完整流程,如图 3-16 所示。

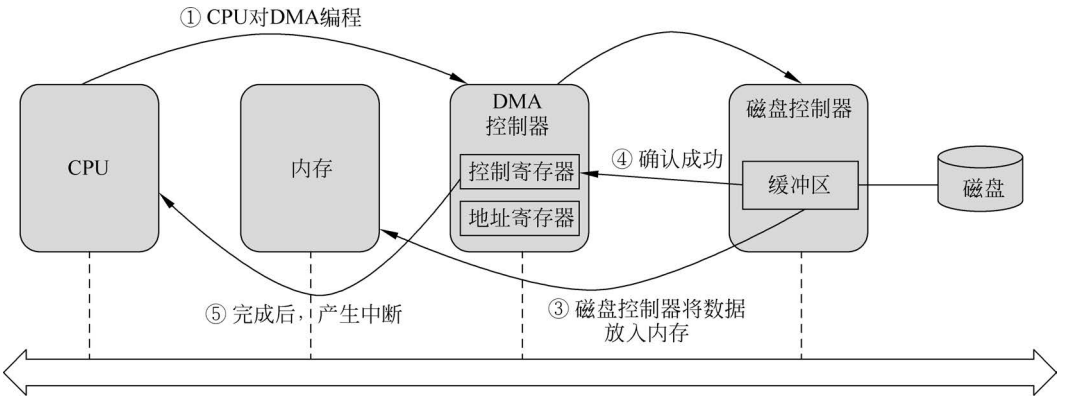


图 3-16 DMA 操作流程

(1) CPU 对 DMA 控制器进行编程,明确指定要传输数据的源地址、目的地址以及数据的长度。此时,CPU 已将 I/O 操作的相关任务全权委托给 DMA 模块,从而能够专注于执行其他计算任务。

(2) DMA 控制器接收到 CPU 发出的 DMA 请求,该请求指示将磁盘数据传送至内存。接收到指令后,DMA 控制器随即启动操作,与磁盘控制器进行交互。

(3) 在 DMA 控制器的管理下,磁盘控制器开始执行数据传送操作,将数据从磁盘读取并直接传送至内存。此过程中,DMA 控制器负责从源地址读取数据,并将其写入指定的目的地址,直至数据传输完毕。

(4) 当数据传输完成后,DMA 控制器会向 CPU 发送一个 DMA 中断信号,以此报告传送操作的结束。在整个 DMA 过程中,DMA 控制器承担了数据传输的主要任务,实现了从源地址到目的地址的直接数据读写,极大地提高了 I/O 操作的效率。

值得注意的是,在 DMA 方式下,处理器无须直接参与数据传输过程,从而能够同时进行其他计算任务,实现了计算与数据传输的并行处理,进一步提升了系统的整体性能。

上面介绍了 DMA 操作的完整流程,其中设备在执行 DMA 操作时,会直接访问物理内存,这带来了安全隐患和多设备共享问题。为了应对这些问题,IOMMU(Input-Output Memory Management Unit,输入输出内存管理单元)应运而生。IOMMU 是一个硬件支持的内存管理单元,专为设备的 DMA 设计,它通过在设备与内存之间添加一个抽象层,实现了对设备访问内存的细粒度控制。

IOMMU 的主要作用有两点:一是地址转换;二是访问控制。如图 3-17 所示,在地址转换方面,IOMMU 可以将设备使用的物理地址转换为机器的物理地址,这允许虚拟机或容器中的设备驱动程序直接使用其自己的物理地址空间,而无须关心实际硬件的物理布局。在访问控制方面,IOMMU 可以确保设备只能访问其被授权的内存区域,从而防止恶意设备或软件对系统内存的非法访问。

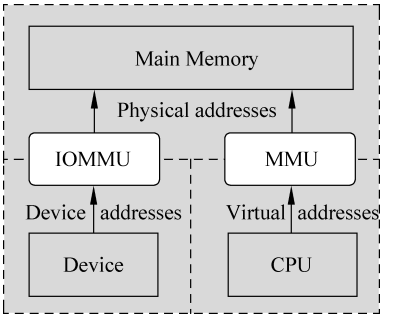


图 3-17　IOMMU 架构

I/O 虚拟化的基本原理主要包括设备发现、访问截获、实现设备的相关功能以及设备共享四个核心部分。下面将结合具体实例,对这些原理进行详细阐述。

(1) 设备发现。

设备发现是 I/O 虚拟化流程的起点,其核心任务在于识别并管理数据中心内所有的物理 IO 设备。鉴于虚拟机无法直接访问物理设备,故需通过 I/O 虚拟化层进行设备的发现与识别。

举例来说,在一个复杂的虚拟化数据中心中,可能存在多种类型的物理设备,如网络适配器、磁盘控制器及串口设备等。I/O 虚拟化层通过扫描硬件总线、解析系统配置信息或运用特定的设备发现协议(如 PCI 设备枚举),实现对这些设备的全面识别。一旦设备被成功发现,I/O 虚拟化层将详细记录其类型、地址、中断号等关键信息,为后续的设备管理与访问提供坚实基础。

(2) 访问截获。

访问截获是 I/O 虚拟化过程中的关键环节。由于虚拟机无法直接与物理 I/O 设备进行交互,I/O 虚拟化层需截获虚拟机对设备的访问请求,并依据预设的配置与策略进行重定向或相应处理。

以虚拟机对磁盘的访问为例,当虚拟机发起磁盘读写请求时,I/O 虚拟化层将迅速截获这些请求。根据虚拟机的具体配置与策略要求,虚拟化层可能选择将请求直接转发至物理磁盘,或将其重定向至虚拟磁盘镜像。若选择重定向至虚拟磁盘,虚拟化层还需负责处理

虚拟磁盘与物理磁盘间的数据同步与映射关系，确保数据的一致性与完整性。

(3) 设备功能实现。

实现设备的相关功能是I/O虚拟化过程中的重要步骤。虚拟化层需精确模拟设备的行为及功能，使虚拟机能够如同访问物理设备般自如地访问虚拟设备。

以网络适配器为例，I/O虚拟化层需精确模拟其数据收发、中断处理及地址解析等功能。当虚拟机发送网络数据时，虚拟化层负责接收并依据网络配置与路由规则将数据转发至目标地址。同时，虚拟化层还需处理来自物理网络适配器的中断与数据包，将其转换为虚拟机可识别的格式，并传递至虚拟机进行处理。

除基本设备功能模拟外，I/O虚拟化层还可提供额外的功能增强。例如，实现网络流量的精细过滤与监控，为虚拟机提供更为安全、可控的网络环境。此外，虚拟化层还支持设备的热插拔与动态配置，极大提升了系统的灵活性与可扩展性。

(4) 设备共享。

设备共享是I/O虚拟化追求的重要目标之一。通过共享物理I/O设备，I/O虚拟化技术能够有效提升资源的利用率并降低成本。

以存储设备为例，传统的物理服务器通常各自配备独立的磁盘阵列。而在虚拟化环境中，多个虚拟机可共享同一物理存储设备。I/O虚拟化层负责管理与调度对这些设备的访问，确保每个虚拟机均能获得充足的存储资源，同时避免资源冲突与性能瓶颈的出现。

设备共享的实现方式多种多样，例如基于存储池的技术将多个物理存储设备整合为单一的逻辑存储池，虚拟机从中动态分配所需存储空间。此外，I/O虚拟化层可以利用磁盘镜像、快照等技术保护数据的完整性与可用性。

然而，设备共享也带来了一定的挑战与限制。例如，如何在不同虚拟机间实现I/O性能的隔离并确保每个虚拟机均能获得公平的访问权限是一个亟待解决的问题。此外，共享设备的安全性亦需得到充分考虑，需要实施严格的访问控制与安全审计机制，以防范潜在的安全风险。

3.4.2 基于软件的I/O全虚拟化

基于软件的I/O全虚拟化技术，其核心在于通过软件层面来模拟与管理I/O设备的行为，这一技术无须特定的硬件支持，因此展现出广泛的适用性与高度的灵活性。在此过程中，设备模型(device model)这一核心概念显得尤为关键。设备模型在基于软件的I/O全虚拟化中发挥着举足轻重的作用，它通过模拟目标硬件设备的接口与行为，为客户呈现一个虚拟设备，使其得以如操作真实设备般自如。

在设备虚拟化过程中，Hypervisor发挥着至关重要的角色。它需要深入研究目标硬件设备的接口定义与内部设计规范，随后以软件的形式模拟这些设备的接口与行为。当虚拟机启动时，Hypervisor所提供的虚拟设备将被虚拟机软件(涵盖BIOS与操作系统)所识别并挂载至虚拟设备总线上。客户机操作系统仅需使用目标设备原有的驱动程序，即可驱动这些虚拟设备。

在设备访问过程中，虚拟机中的驱动程序将发出 I/O 请求并静待设备响应。鉴于 I/O 指令的敏感性，它们将触发 VM-Exit，使得控制权转移至 Hypervisor。随后，Hypervisor 将拦截这些 I/O 请求，并将其传送至相关软件模块进行处理。这些软件模块将模拟这些 I/O 请求，并将 I/O 响应结果反馈至客户机操作系统。只要这些 I/O 响应与真实物理设备的响应保持一致，客户机操作系统便会认为自己正运行在真实的物理硬件平台上。

如图 3-18 所示，设备模型位于 Hypervisor 之中，负责实现设备模拟并处理设备 I/O 请求与响应。

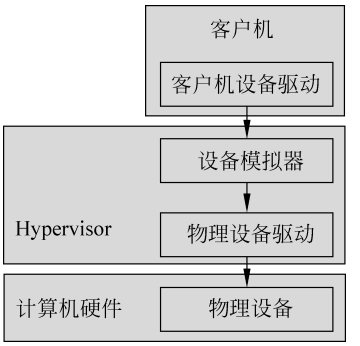


图 3-18　软件模拟虚拟化的结构图

从上述 I/O 虚拟化的流程中可以清晰地看出 I/O 虚拟化对设备模型所提出的主要需求，这主要体现在以下两方面。

1）模拟目标设备软件接口

这一需求是通过软件实现的方式向客户机操作系统提供目标设备的接口，从而使得客户机操作系统能够有效地控制虚拟设备。下面以 3.4.1 节中介绍的端口 I/O(PIO)、MMIO 和 DMA 接口为例，说明 I/O 虚拟化是如何模拟它们的。

(1) PIO。

在 PIO 模式中，设备寄存器的相关 I/O 端口被专设的地址空间所映射。操作系统借助特定的敏感指令，如 x86 架构下的 IN/OUT、INS/OUTS 等，来访问此空间。当客户机执行端口 I/O 操作的访问指令时，会引发 VM-Exit 并触发 Hypervisor 的介入。同时，Hypervisor 会记录所访问的 I/O 端口号、数据宽度及传输方向等关键信息。

在实现 I/O 虚拟化时，对于截获的操作请求，需进行深入的解析和处理。在初始化阶段，设备模型会在 Hypervisor 中预先注册虚拟设备涉及的端口 I/O 及其处理函数，并以数组形式存储处理函数的地址。当客户机执行端口 I/O 操作时，Hypervisor 会根据捕获的 I/O 端口号和数据宽度与已注册的处理函数进行匹配，随后调用相应的函数来模拟设备行为。这些处理函数通过软件实现了设备所需的逻辑功能。

(2) MMIO。

MMIO 作为一种更为普遍的 I/O 方式，尤其适用于寄存器空间需求较大的设备，如网卡和显卡。它与物理内存共享地址空间，使得操作系统能够使用常规内存访问指令(如 MOV)来执行 MMIO 操作。由于内存访问指令并非 MMIO 专用，因此无法像 PIO 那样通过设定敏感指令来截获访问。为了解决这个问题，设备模型通过在每次客户机发起 MMIO 操作时触发缺页异常，导致 VM-Exit，从而使 Hypervisor 能够截获 MMIO 请求并转交给设备模型处理。

(3) DMA。

以 PCI 设备为例，其 DMA 操作由操作系统驱动程序通过访问与 DMA 相关的硬件寄存器实现。驱动程序首先将 DMA 操作的地址写入特定寄存器，随后写入 DMA 命令至另

一寄存器以发起请求。

由于 DMA 的发起依赖于设备寄存器，且驱动程序只能通过端口 I/O 或内存映射 I/O 访问这些寄存器，因此 Hypervisor 能够截获客户机的相关请求，进而实现对 DMA 操作的截获。截获后，Hypervisor 将 DMA 操作转交设备模型进行软件模拟。设备模型无须深入 DMA 的具体实现细节，仅需在客户机分配的内存中进行数据的读写操作。

为实现这一过程，设备模型利用 Hypervisor 的内存管理模块，将客户机用于 DMA 的内存空间映射至自身地址空间。一旦接收到 I/O 请求，设备模型便发起系统调用，以 DMA 方式读写映射的内存。数据传输完成后，设备模型通过虚拟中断控制器向虚拟机注入虚拟设备中断，从而结束 DMA 操作。

2）实现目标设备功能

除了提供虚拟设备的 I/O 访问接口外，完整的 I/O 虚拟化还需确保实现虚拟机所期望的设备功能，以确保在虚拟机发起 I/O 操作后能够返回准确无误的结果。在功能实现过程中，我们无须严格遵循目标物理设备的硬件结构和模块组成，这使得虚拟设备功能的实现更为灵活多变。

以 IDE（Integrated Drive Electronics，集成磁盘电子接口）存储系统为例，真实设备由 IDE 控制器和集成的 IDE 硬盘共同构成。其中，IDE 控制器作为一个包含可控软件接口的 PCI 设备，而硬盘本身则由控制器进行管控，并不具备独立的访问接口。因此，在虚拟 IDE 时，仅需模拟 IDE 控制器的软件接口并供给虚拟机使用即可。此外，根据应用的具体需求和系统架构的特点，还可以在虚拟设备中引入一些真实物理设备所不具备或难以实现的功能及特性（例如虚拟 IDE 的增量存储和备份等），从而进一步丰富和拓展虚拟设备的功能和应用范围。

3.4.3 I/O 半虚拟化

I/O 半虚拟化技术是在全虚拟化技术的基础上发展而来的。全虚拟化技术通过软件模拟的方式实现虚拟机对物理 I/O 设备的访问，但这种方式通常会带来较大的性能开销。为了解决这个问题，I/O 半虚拟化技术应运而生。

I/O 半虚拟化技术的核心思想是通过虚拟机与虚拟化层的紧密合作实现高效的 I/O 通信。它利用虚拟机对虚拟化环境的感知能力，通过一组优化的接口和协议，减少虚拟化层在 I/O 操作中的开销。

具体来说，I/O 半虚拟化技术通常包括以下几个关键步骤。

（1）接口定义与协商。

虚拟机与虚拟化层之间通过定义一组标准的 I/O 接口和协议进行通信。这些接口和协议通常包括设备发现、初始化、配置、数据传输等功能。虚拟机在启动时，会与虚拟化层进行接口协商，确定双方支持的 I/O 操作方式和参数。

（2）前端与后端的协同工作。

在 I/O 半虚拟化技术中，虚拟机内部通常包含一个或多个前端驱动，负责处理虚拟机

的I/O请求；而虚拟化层则提供相应的后端服务，负责与物理设备进行交互。前端驱动和后端服务之间通过共享内存、消息队列等高效通信机制进行连接和协同工作。

（3）请求处理与数据交换。

当虚拟机发起I/O请求时，前端驱动会将请求封装成虚拟化层可识别的格式，并通过通信机制发送给后端服务。后端服务接收到请求后，会解析并执行相应的操作，例如与物理设备进行交互或进行必要的转换。处理完成后，后端服务会将结果数据回传给前端驱动，由前端驱动交付给虚拟机的操作系统或应用程序。

上面介绍了I/O半虚拟化技术的基本原理，接下来将聚焦I/O虚拟化领域的典型代表——virtio技术。

virtio作为I/O半虚拟化技术的典范，确立了一套标准化的I/O虚拟化接口与协议，旨在实现虚拟机和虚拟化层之间的高效I/O通信。

virtio的架构主要由三部分构成：前端驱动、virtio后端以及virtio设备。如图3-19所示，前端驱动部署于虚拟机内部，负责将虚拟机的I/O请求转换为virtio特有的格式，并通过共享内存或其他通信机制传送至后端。virtio后端则运行于虚拟化层，负责接收前端发送的请求，与物理设备进行交互，并将结果数据反馈至前端。而virtio设备作为虚拟机内部的一个虚拟设备，为前端驱动提供标准的I/O接口，使虚拟机能够如同访问物理设备般轻松访问它。

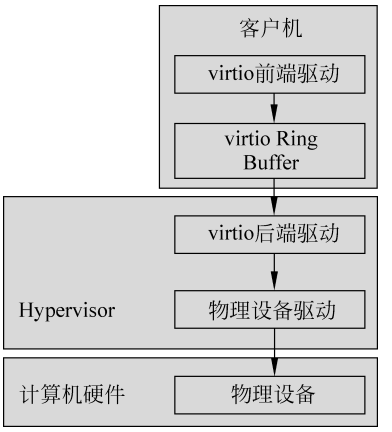


图3-19　virtio半虚拟化架构

（1）virtio的通信机制。

virtio采用了一种基于共享内存与事件驱动的通信机制。前端驱动与后端之间通过共享内存进行数据交换，有效避免了不必要的数据复制开销。同时，virtio还利用事件机制实现双方的同步与通知。当后端处理完毕I/O请求或发生其他事件时，它会通过事件通知前端进行相应的处理。

（2）virtio的性能优势。

由于virtio采用了优化的通信机制和接口设计，它显著减少了虚拟化层在I/O操作中的开销，从而提升了I/O性能。相较于传统的全虚拟化技术，virtio能够提供更高的吞吐量和更低的延迟，使虚拟机在I/O性能方面更加接近物理机。此外，virtio还展现出卓越的可扩展性与灵活性。它支持多种类型的I/O设备，涵盖网络、存储等领域，并可根据具体需求进行定制与扩展。这使得virtio能够适应不同场景下的I/O虚拟化需求。

（3）virtio的应用与生态。

目前，virtio已得到广泛的应用与支持。众多主流虚拟化平台均集成了virtio技术，并提供了相应的前端驱动与后端实现。此外，virtio还得到了开源社区与企业的积极参与和支持，形成了一个庞大的生态系统，这为virtio的发展与应用提供了强大的动力与支持。

3.4.4 硬件辅助的 I/O 虚拟化

传统的 I/O 虚拟化通常依赖于软件层面的模拟和拦截，这种方式虽然能够实现基本的 I/O 功能，但往往伴随着较大的性能损耗。而硬件辅助的 I/O 虚拟化技术则通过在硬件（如 CPU、I/O 控制器等）中内置虚拟化支持，实现了对虚拟机 I/O 请求的直接转发和处理，从而大幅减少了虚拟化带来的性能开销。

硬件辅助的 I/O 虚拟化通常依赖于处理器、I/O 设备以及虚拟化平台之间的紧密协作。例如，处理器可能提供特定的虚拟化指令集，以加速虚拟机的 I/O 操作；I/O 设备可能支持直接内存访问技术，允许虚拟机直接访问设备的内存缓冲区；而虚拟化平台则负责管理和调度这些资源，确保虚拟机之间的隔离性和安全性。

如图 3-20 所示，无须设备模拟器或者前后端驱动，Hypervisor 中的物理设备驱动与物理设备配合实现 I/O 虚拟化，支持客户机中的客户机设备驱动以原生方式使用虚拟化后的 I/O 设备。

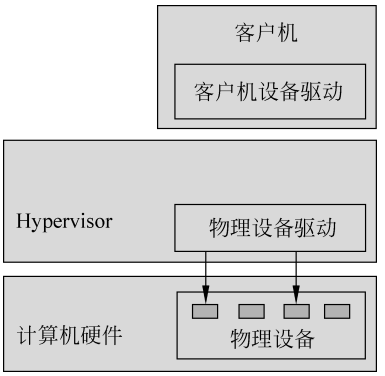


图 3-20 I/O 硬件虚拟化架构

通过硬件辅助的 I/O 虚拟化技术，数据中心可以实现更高的 I/O 吞吐量、更低的延迟以及更好的资源利用率。这有助于提升虚拟机的性能表现，满足各种应用场景的需求。硬件辅助的 I/O 虚拟化主要分为设备直通和 SR-IOV 两类。

1）设备直通

设备直通（Pass-Through）是硬件辅助 I/O 虚拟化技术的一种实现方式，它允许虚拟机直接访问物理 I/O 设备，而无须经过虚拟化层的干预。这种方式极大地提升了虚拟机的 I/O 性能，使得虚拟机能够像物理机一样高效地访问硬件设备。

以一个网络适配器的设备直通为例，其实现过程如下。

（1）配置阶段：在虚拟化平台上，管理员选择将某个物理网络适配器设置为直通模式，并将其分配给特定的虚拟机。这个过程通常涉及对虚拟化平台和物理设备的配置。

（2）设备映射：虚拟化平台在配置完成后，会建立虚拟机与物理网络适配器之间的直接映射关系。这种映射关系确保了虚拟机能够直接访问到分配给它的物理网络适配器。

（3）虚拟机访问：当虚拟机启动时，它会识别到分配给它的物理网络适配器，并像访问普通网络设备一样进行配置和使用。虚拟机发出的网络请求将直接通过物理网络适配器发送出去，无须经过虚拟化层的处理。

通过设备直通，虚拟机可以获得接近物理机的网络性能，适用于需要高性能网络吞吐量的应用场景，如大数据处理、云计算等。

然而，设备直通也存在一些限制。首先，它通常要求物理设备支持虚拟化感知功能，否则可能无法实现直通效果。其次，由于设备被直接分配给虚拟机使用，因此无法实现设备的共享，这可能导致资源利用率不高。此外，设备直通还可能增加管理的复杂性，因为管理

员需要手动配置和管理每个虚拟机的设备分配。

2）SR-IOV

SR-IOV(Single Root I/O Virtualization)是另一种重要的硬件辅助I/O虚拟化技术。它通过在物理I/O设备上创建多个虚拟功能，使得每个虚拟机可以独立地访问一个或多个虚拟功能，从而实现I/O资源的共享和隔离。

下面介绍物理功能(Physical Function，PF)和虚拟功能(Virtual Function，VF)这两个重要概念。

物理功能(PF)：用于支持SR-IOV功能的PCI功能，如SR-IOV规范中定义。PF包含SR-IOV功能结构，用于管理SR-IOV功能。PF是全功能的PCIe功能，可以像其他任何PCIe设备一样进行发现、管理和处理。PF拥有完全配置资源，可以用于配置或控制PCIe设备。

虚拟功能(VF)：与物理功能关联的一种功能。VF是一种轻量级PCIe功能，可以与物理功能以及与同一物理功能关联的其他VF共享一个或多个物理资源。VF仅允许拥有用于其自身行为的配置资源。

SR-IOV技术的核心在于物理I/O设备支持虚拟功能的创建和管理。物理设备通过PCI Express接口与宿主机相连，并在宿主机上配置为支持SR-IOV模式。在配置完成后，物理设备会创建多个虚拟功能，并为每个虚拟功能分配独立的资源(如中断、DMA通道等)。

如图3-21所示，以SR-IOV网卡为例，物理网卡(Network Interface Card，NIC)通过SR-IOV技术创建多个虚拟网卡，并直通给虚拟机使用。

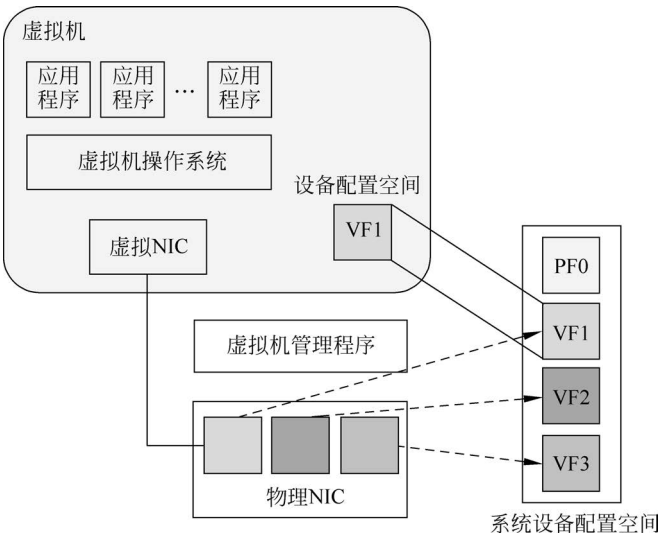


图3-21　SR-IOV网卡示例

虚拟机在创建时，可以选择绑定到一个或多个虚拟功能。一旦绑定完成，虚拟机就可以像操作物理设备一样操作这些虚拟功能。由于虚拟功能具有独立的资源，因此不同虚拟机之间的I/O操作不会相互干扰，从而实现了I/O资源的隔离。

SR-IOV技术相比传统的I/O虚拟化方法具有更高的性能优势。由于虚拟机直接访问

虚拟功能的硬件资源，避免了在软件层进行 I/O 操作的转换和模拟，因此可以获得更高的吞吐量和更低的延迟。这使得 SR-IOV 技术在高性能网络、存储等应用场景中具有广泛的应用前景。

在具体厂商实现方面，许多主流的处理器和 I/O 设备厂商都提供了 SR-IOV 的支持。例如，Intel 的处理器和网卡、Mellanox 的网络适配器等都支持 SR-IOV 功能。此外，虚拟化平台如 VMware、Red Hat 和 Microsoft 等也提供了对 SR-IOV 的集成和支持。

在使用 SR-IOV 时，管理员需要确保所使用的物理设备支持 SR-IOV 功能，并在虚拟化平台上进行相应的配置和启用。同时，还需要注意设备的兼容性和稳定性问题，以确保虚拟机的正常运行和性能表现。

尽管硬件辅助的 I/O 虚拟化技术带来了显著的性能提升和灵活性增强，但它们也面临着一些挑战。首先，硬件辅助的 I/O 虚拟化技术通常要求设备支持特定的虚拟化功能，这可能导致设备选择的局限性。其次，配置和管理这些技术需要较高的技术水平和经验，对管理员的要求较高。最后，随着技术的不断发展，如何保持与最新硬件和平台的兼容性也是一个需要解决的问题。

3.5 GPU 虚拟化

随着音视频类应用的广泛普及，高性能计算和图形渲染领域对图形绘制的需求日益旺盛。然而，传统的 CPU 作为计算机系统的核心运算与控制单元，已逐渐无法满足这一日益增长的需求。因此，急需一种专门用于处理图形的核心处理器来填补这一空白。在此背景下，NVIDIA 公司于 1999 年推出了具有划时代意义的 Geforce256 图形处理芯片，并首次提出了图形处理器的概念。自此以后，NVIDIA 显卡的芯片便以 GPU 命名，标志着图形处理进入了一个全新的时代。

GPU 的引入显著减轻了计算任务对 CPU 的依赖，并承担了原本由 CPU 处理的部分工作负载，尤其在三维图形处理方面展现出了卓越的性能。近年来，随着技术的不断进步，GPU 在渲染、编解码和计算等领域的应用愈发广泛，对个人计算机游戏和电子商务市场的发展起到了巨大的推动作用。

同时，随着智能化时代的来临，GPU 的应用范围进一步拓展至人工智能领域。商业公司和政府机构对人工智能技术的关注和使用迅速增加，纷纷搭建自有智能平台以赋能各项业务。在这一过程中，如何基于采购的 GPU 服务器构建高效、弹性的 GPU 资源池，以满足不断变化的计算需求，成为一个亟待解决的关键问题。

3.5.1 GPU 虚拟化基本原理

GPU 虚拟化技术旨在将物理 GPU 资源抽象为多个虚拟 GPU（vGPU），以供多个虚拟机或容器同时使用。通过这种方式，不仅提高了 GPU 资源的利用率，还增强了系统的灵活性和可扩展性。与 CPU 和内存虚拟化类似，GPU 虚拟化也需要解决资源隔离、性能分配和

管理复杂性等问题。

根据实现方式的不同，GPU 虚拟化技术可以分为基于软件的 GPU 虚拟化和基于硬件的 GPU 虚拟化两大类。基于软件的 GPU 虚拟化主要通过 GPU 厂商提供的专门用于 GPU 虚拟化的驱动程序来实现 GPU 资源的共享和隔离，例如 NVIDIA GRID vGPU 和 Intel GVT-g 技术。而基于硬件的虚拟化，如 NVIDIA 的 Multi-Instance GPU（MIG）技术，则直接在 GPU 硬件级别支持资源的划分和隔离。本节将分别对这两种 GPU 虚拟化的实现方式进行简要阐述，并介绍它们所对应的业界主流产品。

3.5.2　基于软件的 GPU 虚拟化

基于软件的 GPU 虚拟化技术的实现原理主要依赖于时分复用机制。GPU 的时分复用与 CPU 在进程间的时分复用概念相似。简单而言，GPU 时分复用是指将一个 GPU 的使用时间按照特定的时间段进行分片，每个虚拟机在分配到的特定时间片内独享 GPU 的全部硬件资源。目前，关于 GPU 时间片的轮转主要有两种机制。

（1）在当前 GPU 上下文的 BatchBuffer/CMDBuffer 执行结束后启动调度，并将 GPU 的控制权交给下一个时间片的所有者。

（2）在特定时间片结束时进行严格切换，即使当前 GPU 执行尚未完成也会被强行中断，并将控制权转交给下一个时间片的所有者。这种方式确保了 GPU 资源在不同虚拟机之间的平均分配。

然而，GPU 时分复用面临着多任务干扰的问题。在实际应用中，多个任务对 GPU 利用率和显存利用率之和往往远低于理想状态，同时单个任务的完成时间（Job Completion Time，JCT）也会显著增加。这主要是由以下几个因素造成的。

（1）计算碰撞：当多个任务同时需要使用 GPU 资源进行计算时，它们之间会发生竞争和冲突，导致每个任务的完成时间延长。即使某个任务在 GPU 切换到其他任务时正在进行 CPU 计算、I/O 操作或通信，但当它需要 GPU 资源时，如果时间片能够立即切换回该任务，那么理论上不会对它的完成时间产生影响。但在实际场景中，多个任务往往同时需要 GPU 资源，从而加剧了计算碰撞的问题。

（2）通信碰撞：当多个任务同时需要使用显存带宽进行数据传输时（例如在主机内存和 GPU 显存之间），它们之间也会发生竞争和冲突。这种通信碰撞会进一步降低 GPU 的利用率和性能表现。

（3）GPU 上下文切换耗时长：与 CPU 上下文切换相比（通常在纳秒级别），GPU 的上下文切换时间要长得多，可能达到几百纳秒甚至毫秒级别。这意味着频繁的 GPU 调度操作会带来较大的开销和性能损失。例如：按照 2ms 的时间片进行 GPU 调度，每次调度的上下文切换大概花费 0.5ms，那么应用任务在理论上只能使用约 80％的 GPU 资源。

根据图 3-22 所示的 GPU 时分复用原理，应用程序需通过调度器来获取 GPU 的时间片资源。在成功获取的时间片内，应用程序将执行计算任务；一旦时间片消耗殆尽，应用程序则需等待下一次的时间片分配。在 GPU 时分复用的技术框架内，进一步细分为 SR-IOV 和 MDEV 两种实现方案。

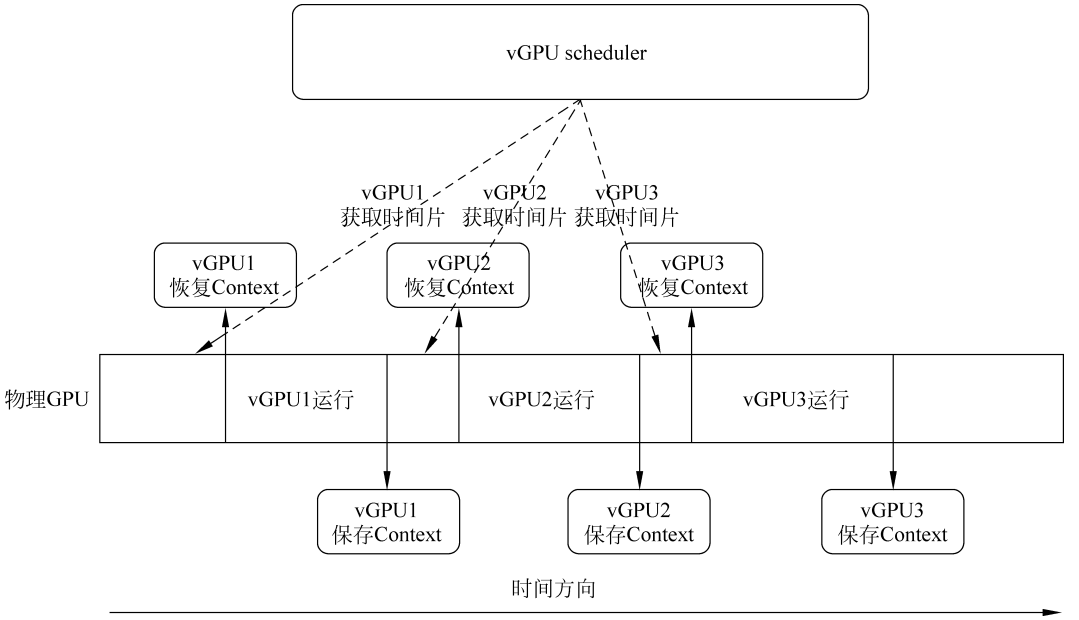


图 3-22 GPU 时分复用原理

1) SR-IOV

GPU 作为一种典型的 PCIe(Peripheral Component Interconnect Express)设备，其 SR-IOV(SR-IOV 的原理参考 3.4.4 节)规范的实现为虚拟机提供了更为高效和灵活的图形处理能力。如图 3-23 所示，根据 SR-IOV 规范，GPU 可以被切分为多个物理功能(PF)或虚拟功能(VF)，随后这些功能通过 PCI 设备直通技术被分配给多个 VM 使用。在此过程中，每个 VF 都被赋予了独立的 Bus/Slot/Function 号，确保其在系统中的唯一性。

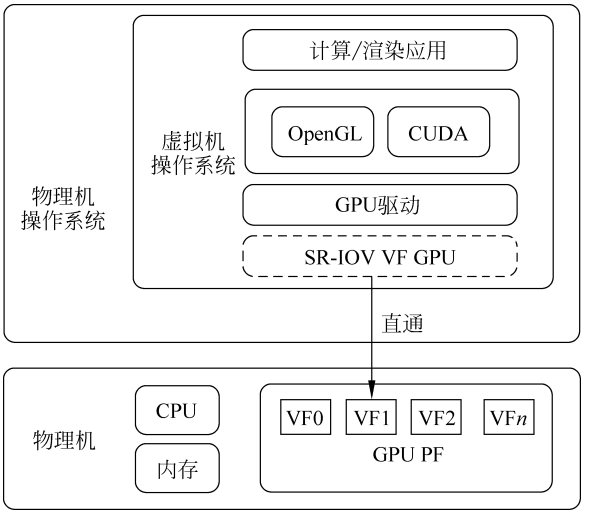


图 3-23 GPU 的 SR-IOV 直通架构

当IOMMU(IOMMU的原理参考3.4.1节)与VT-d(Intel Virtualization Technology for Directed I/O)接收到来自这些VF的DMA(DMA的原理参考3.4.1节)请求时,它们能够通过查询IOMMU转换表(IOMMU Translation Table),从而实现从Guest Physical Address(GPA)到Host Physical Address(HPA)的地址转换。这种机制确保了虚拟机在访问物理硬件资源时的正确性和高效性。

然而,值得注意的是,GPU的SR-IOV实现并不仅仅局限于PCIe事务层(TLP)的VF路由标识封装。尽管这种封装有助于规避运行时(runtime)的软件DMA翻译需求,但硬件层面的支持同样至关重要。以AMD S7150 GPU为例,虽然它表面上支持SR-IOV规范,但实际上硬件仅在PCIe层面对VF进行了抽象处理。因此,在主机(Host)端,还需要一个具备虚拟化感知能力(Virtualization-Aware)的物理GPU(pGPU)驱动程序来负责VF的模拟和调度工作。

与此同时,NVIDIA在其Ampere架构及以后的GPU产品中也开始支持SR-IOV方案。这一举措表明,随着虚拟化技术的不断发展和完善,越来越多的GPU厂商开始重视并投入资源来优化其在虚拟化环境中的性能表现。

2) MDEV

Mediated Passthrough是一种完全基于软件定义的GPU虚拟化解决方案。其核心思想在于将与GPU性能直接相关的访问请求直接传递给虚拟机,而将那些与性能无直接关联的功能性访问请求在MDEV模块中进行模拟处理。目前,此领域的典型产品包括NVIDIA GRID vGPU和Intel GVT-g(如KVMGT、XenGT)。下面以NVIDIA GRID vGPU产品为例,说明基于MDEV方案的GPU虚拟化产品的基本架构。

如图3-24所示,NVIDIA GRID vGPU产品的架构大致如下。

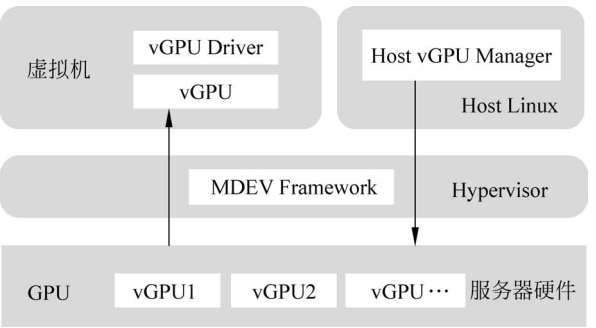


图3-24　GRID vGPU架构图

(1) Host上安装了NVIDIA GRID Host驱动程序,它会在Host上运行一个Host vGPU Manager服务,该服务专门负责管理虚拟图形处理器(vGPU)。

(2) 当用户需要使用vGPU时,Host vGPU Manager服务会将物理NVIDIA GPU切割成多个vGPU实例,并通过PCI直通方式将这些实例挂载到虚拟机中。

(3) 在虚拟机内部需要安装与Host驱动程序配套的NVIDIA vGPU驱动程序。该驱动程序与Host上的驱动程序通过MDEV Framework通信,使得虚拟机能够充分利用分配

的 vGPU 资源。

NVIDIA GRID vGPU 在实现了 GPU 时分复用的基础上，进一步支持以下三种调度器模式，用于满足不同的 GPU 虚拟化需求。

(1) Fixed Share：此调度模式确保共享同一块 GPU 的所有 vGPU 获得均等的性能分配。通过预先定义的资源分配，每个 vGPU 都能获得稳定的性能表现。

(2) Best Effort：在此模式下，采用轮询(round-robin)调度算法，使所有 vGPU 根据实际需求共享 GPU 资源，以实现资源利用的最优化。此调度器确保在有空闲的 vGPU 时，其对应的资源能够被充分利用，从而提高整体资源利用率。

(3) Equal Share：该模式为每个运行的 vGPU 分配相同的 GPU 资源。当 vGPU 的数量增加或减少时，每个 vGPU 所能分配到的资源会相应变化，从而导致其性能表现也发生调整。

需要注意的是，Best Effort 是系统的默认调度器。其实不论采用何种调度器，本质上都是通过硬件调度器实现时间分片，确保每个 vGPU 都能获得执行自身程序所需的时间片。以上三种调度器模式为 NVIDIA GRID vGPU 提供了灵活的资源管理方式，以满足不同虚拟化场景下的需求。

3.5.3 基于硬件的 GPU 虚拟化

NVIDIA 自 Ampere 架构起引入了 Multi-Instance GPU(MIG)技术，该技术能够在硬件层级上对 GPU 资源进行水平切分，进而实现硬件资源的隔离与故障隔离，成为当前隔离性最优的方案。值得一提的是，NVIDIA MIG 是现今唯一支持硬件隔离的方案。

Ampere 架构通过独特的硬件设计，使 GPU 具备了创建子 GPU(GI)的能力。这些子 GPU 在计算、内存带宽、故障隔离、错误处理及恢复等方面均保持相对独立，从而确保了较高的服务质量(QoS)。MIG 技术的核心原理在于对物理卡上可用的物理资源进行分块与再组合，这些资源涵盖系统通道、控制总线、算力单元(TPC)、全局显存、L2 缓存以及数据总线等。经过分块后的资源将被重新组合，以确保每个子 GPU 都能实现数据保护、故障隔离独立性以及服务的稳定性。

然而，MIG 技术也存在一定的局限性，主要表现在高成本和不灵活性两方面。具体而言，该技术仅支持高端 GPU，并且仅限于 CUDA 环境。此外，每款 GPU 所支持的实例数量也相对较少。

以 A100 GPU 为例，其在硬件底层被拆分为 7 个子 GPU 实例，每个实例均拥有独立的核心、缓存和内存。这种设计不仅满足了数据中心对 GPU 资源进行分割的需求，还能在同一块显卡上并行运行不同的训练任务，从而大幅提升资源的利用率和灵活性。如图 3-25 所示，将拆分出的拥有独立硬件资源的实例采用 GPU 直通方式直通进虚拟机，从而使虚拟机能够拥有基本无损耗的物理 GPU 性能。

MIG 方案的关键在于对 GPU 物理资源的切分，切分出的每个子 GPU 等同于一个物理 GPU。其包含的概念如下。

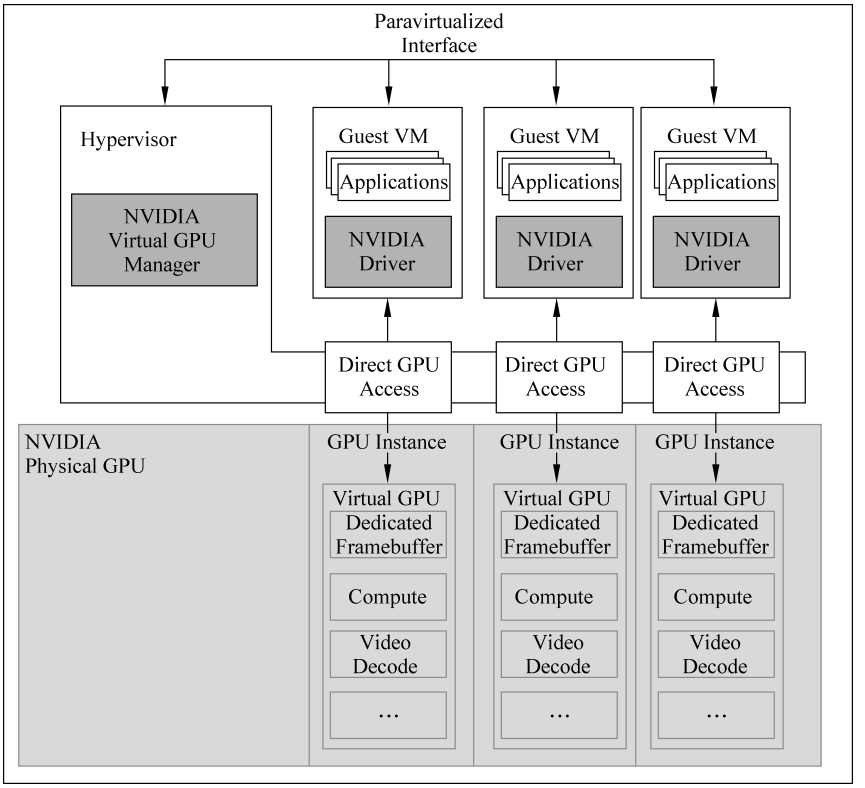


图 3-25　MIG 方案架构图

(1) 流处理器 SM：SM 切片由 GPU 决定，如 A100 有 108 个 SM 单元，14 个 SM 单元为一个切片，所以 A100 有 7 个切片。一个 GPU 实例最少需要一个切片。

(2) 显存：A100 有 40GB 显存，5GB 为一个单元，最多分割为 8 个单元，可以组合为 10GB、20GB、40GB 这些规格。

(3) L2 Cache、系统通道等：MIG 根据 SM 和显存分块和组合。

(4) GPU 引擎：GPU 引擎是在 GPU 上执行工作的引擎。最常用的发动机是执行计算指令的计算/图形引擎。其他引擎包括负责执行 DMA 的复制引擎(CE)、用于视频解码的 NVDEC、用于视频编码的 NVENC 等。每个引擎可以独立调度并执行不同的工作 GPU 上下文。

(5) GPU Instance(GI)：在物理 GPU 上将需要的分块好的单元选取出来组合成的 GPU 实例，GI 之间 SM、显存、引擎等资源都是隔离的。

(6) Compute Instance(CI)：GI 可以细分为多个 CI，CI 算力独立，包含 GI 的 SM 切片和其他引擎的子集，共享显存和引擎。GI 也可以作为一个整体使用。

(7) MIG Profile：每款 MIG GPU 都有配置文件，按照配置文件上的规格分割。

NVIDIA MIG 技术为 GPU 虚拟化领域的发展开辟了新的途径。该技术有效地解决了传统 GPU 虚拟化方案中面临的性能损失与资源争用难题，实现了真正的硬件级别隔离。

随着技术的不断革新与进步，更多的 GPU 制造商也在投入类似的硬件隔离技术研发之中，推动硬件隔离技术的不断演进与发展。

3.5.4 方案对比及总结

3.5.2 节和 3.5.3 节已经介绍了多种 GPU 虚拟化的实现方案，表 3-1 是这些方案之间的对比分析。

表 3-1 MIG、MDEV 和 SR-IOV 对比说明

对 比 项	MIG	MDEV	SR-IOV
切分方式	物理隔离	时分复用	时分复用
支持模式	计算	计算/渲染	计算/渲染
支持系统	Linux	Windows/Linux	Windows/Linux
使用方式	物理机/虚拟机	虚拟机	虚拟机
性能损耗	5%左右	1 切 1 在 5%以内，1 切多在 10%～30%	1 切 1 在 5%以内，1 切多在 10%～20%
支持 GPU 的类型	A100、A30、H100	NVIDIA 的计算卡	NVIDIA Ampere 架构及以后
切分规格	支持不同规格组合切分	同 GPU 同一规格	同 GPU 同一规格

对比不同方案的优缺点和适用场景，有助于更好地理解各种方案之间的差异和选择依据。在选择 GPU 虚拟化方案时，需要根据具体的应用场景、性能需求、资源利用率和成本等因素进行综合考虑。未来随着技术的不断发展，我们期待出现更多创新性的 GPU 虚拟化方案，以满足不断增长的计算需求。

GPU 虚拟化构成了实现 GPU 资源池化的基石和关键技术，而 GPU 池化则可以被视为 GPU 虚拟化在资源管理层面的进阶应用，它作为一种重要的解决策略，有效应对了传统 GPU 资源分配方式所面临的挑战。

实施 GPU 池化能够显著提升 GPU 资源的利用效率和管理便捷性，进而为云计算、大数据处理以及人工智能等关键领域的发展提供坚实的技术支撑。在本书的第 8 章 AI 算力池化技术中，将进一步深入探讨与 GPU 池化相关的内容，并介绍其实现方式与应用场景。

3.6 计算虚拟化实践

在前面的章节中，已经对计算虚拟化的各个组成部分的基本原理和具体实现进行了简要阐述。华为 DCS 的计算虚拟化平台以 QEMU-KVM 开源系统为基础，并在此之上发展出了自有的虚拟化解决方案。接下来，将结合 DCS 解决方案，给出计算虚拟化的一些实践应用。

QEMU-KVM 是一种基于 Linux 内核的开源虚拟化系统，其中 QEMU（Quick EMUlator）是一个通用的机器模拟器和虚拟化器，能够模拟多种不同的处理器架构和设备，而 KVM

(Kernel-based Virtual Machine)则是Linux内核中的一个模块,提供了对CPU和内存的虚拟化支持。将两者结合使用,可以实现高效的虚拟化环境。

如图3-26所示,在QEMU-KVM虚拟化系统中,KVM负责CPU和内存的虚拟化工作。它利用Linux内核中的虚拟化扩展功能,将物理CPU和内存资源抽象成虚拟资源,并通过特定的接口暴露给QEMU使用。QEMU则负责虚拟机的创建和管理,它通过解析虚拟机的配置文件,加载虚拟机的镜像文件,并启动虚拟机运行。同时,QEMU还负责模拟虚拟机的I/O设备,使得虚拟机能够像访问物理设备一样访问虚拟设备。

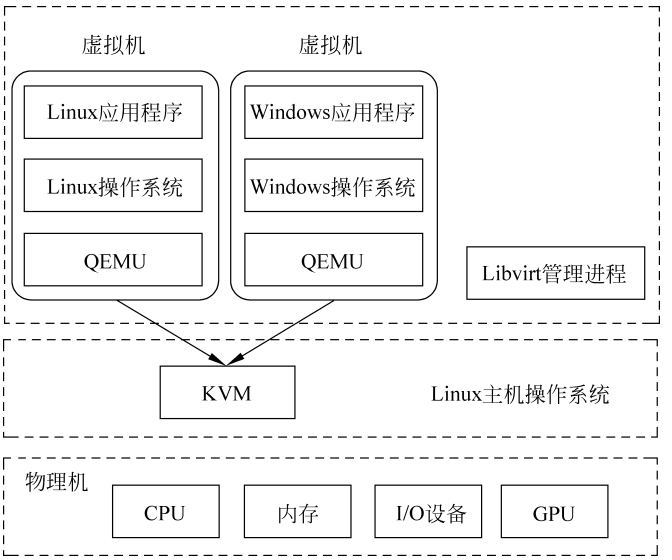


图3-26　QEMU-KVM虚拟化系统

QEMU-KVM虚拟化系统具有以下几个显著优势。

(1) 它充分利用了Linux内核的虚拟化扩展功能,实现了高效的CPU和内存虚拟化,使得虚拟机能够获得接近物理机的性能表现。

(2) QEMU作为一个通用的虚拟化软件,支持多种不同的处理器架构和设备模拟,使得用户可以在同一平台上运行多种不同的操作系统和应用程序。

(3) QEMU-KVM还具有强大的扩展性和灵活性。用户可以根据需要自定义虚拟机的配置和性能参数,满足不同的业务需求。同时,QEMU-KVM还提供了丰富的管理工具和API接口,方便用户进行虚拟机的监控和管理。

DCS提供的计算虚拟化技术,在QEMU、KVM以及Libvirt等技术架构的基础上,针对CPU、内存和I/O等虚拟化领域,进行了深入的扩展与创新,从而衍生出多样的计算虚拟化能力。

3.6.1　CPU服务质量保证

DCS的虚拟机的CPU服务质量保证(CPU QoS)用于保证虚拟机的计算资源分配,隔

离虚拟机间由于业务不同而导致的计算能力相互影响，满足不同业务对虚拟机计算性能的要求，最大程度复用资源，降低成本。

在创建虚拟机时，可根据虚拟机预期部署业务对 CPU 的性能要求而指定相应的 CPU QoS。不同的 CPU QoS 代表了虚拟机不同的计算能力。基于 vCPU 的时分复用原理，通过指定的 CPU QoS 配置项计算每个 vCPU 的份额并加以设置，进而实现对 vCPU 计算能力的最低保障和资源分配的优先级。

当前 DCS 提供三个维度的配置，来保证虚拟机 CPU 计算资源的服务质量：CPU 预留、CPU 份额、CPU 上限。

(1) CPU 预留：指一个虚拟机各个 vCPU 在竞争物理 CPU 资源时至少占用的 CPU 主频大小，用于保证 vCPU 的最低计算能力。

(2) CPU 份额：指一个虚拟机在竞争物理计算资源的能力大小总和，是一个相对值。虚拟机获得的 CPU 计算资源，是与其他虚拟机 CPU 份额按相对比例瓜分物理 CPU 除预留外的可用计算资源。

(3) CPU 上限：指一个虚拟机各个 vCPU 在竞争物理 CPU 资源时最多占有的物理 CPU 能力的大小。用于设置虚拟机的计算能力上限。

CPU 服务质量保证的执行流程如下。

(1) 如图 3-27 所示，DCS 会在每个计算节点上启动一个 CPU QoS 进程。

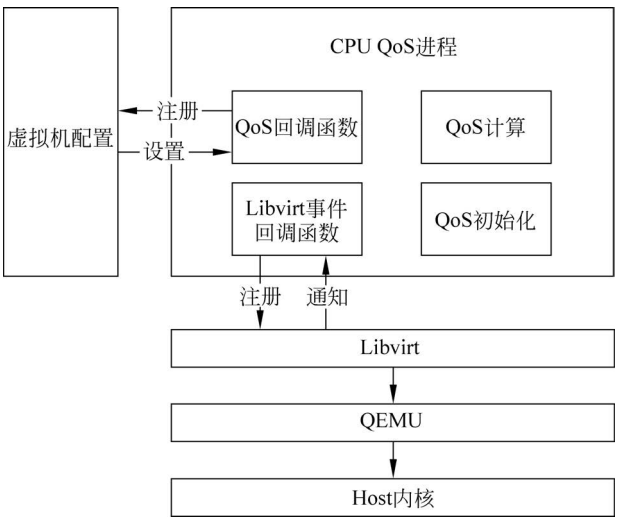


图 3-27 CPU 服务质量保证执行流程示意图

(2) 在创建虚拟机时，配置虚拟机的 CPU QoS 参数。

(3) 在启动虚拟机时，CPU QoS 进程会通过虚拟机配置获取 CPU QoS 的配置参数，然后通过自研 QoS 算法进行 QoS 计算。

CPU QoS 进程在计算完成后将 QoS 值设置到宿主机内核中，实现对 vCPU 调度时间片的控制。

3.6.2　智能内存复用

DCS 的智能内存复用技术是在虚拟化平台下对虚拟机所用内存的一种管理技术，可以使用户在主机上运行超过物理内存总量的虚拟机，提升虚拟机的部署密度，达到节省成本的目的。

内存复用的自动化管理组件是一个守护进程，在后台根据当前空闲的物理内存，综合运用内存气泡、内存交换和内存合并三种技术，自动控制每个虚拟机的可用内存量，从而提升虚拟机的密度。

（1）内存气泡：在虚拟机运行过程中，由虚拟机中的内存气泡前端驱动结合 QEMU 中的后端驱动来动态占用或释放内存，从而动态改变这台虚拟机的当前可用内存。

（2）内存交换：把虚拟机中不活跃的内存数据交换到主机的交换空间，释放内存资源给其他虚拟机用，从而提高内存复用率。

（3）内存合并：多台虚拟机可能有很多内存页内容是完全相同的，将这些内容完全相同的内存页合并，只在物理内存上保留一个副本，将那些相同的内存页都指向该副本。在发生内存写操作时，将合并页打散，重新申请一个内存页，存放新写入的内存。

智能内存复用的执行流程如下。

（1）如图 3-28 所示，内存复用会启动一个守护进程作为管理组件。当检测到主机上所有虚拟机的内存规格之和大于主机上虚拟化域的总物理内存时，进入内存复用状态。

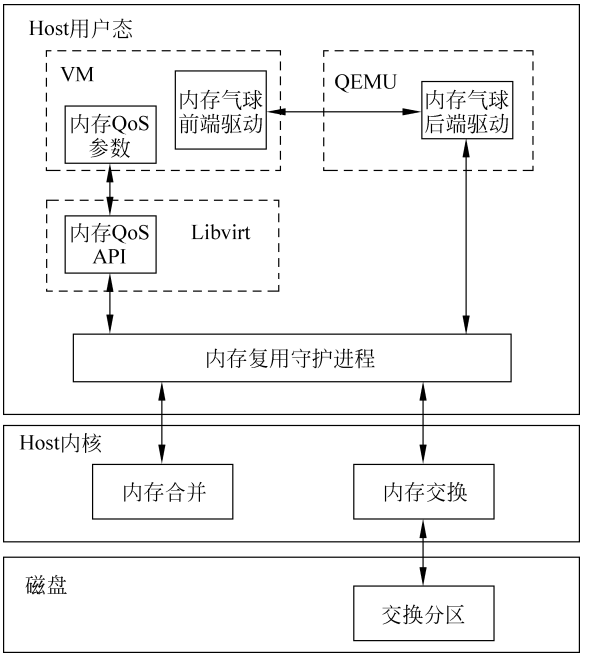


图 3-28　内存复用架构图

（2）进入内存复用状态后，内存复用策略周期性探测所有虚拟机内部的真实内存压力，对于保持了一段时间低压力的虚拟机，内存气泡技术可以将它们的内存释放出来。

（3）当所有的虚拟机真实使用内存大于虚拟化域总物理内存时，内存复用策略会触发内存交换。

（4）当剩余内存和交换分区都接近用完时，内存复用策略会暂停虚拟机。之后如果剩余内存增加到一定程度，内存复用会提高虚拟机的内存上限并恢复虚拟机。

3.6.3 虚拟机热迁移

虚拟机运行在物理机上时，物理机会存在资源分配不均（如负载过重、负载过轻）、物理服务器硬件更换、软件升级、组网调整、故障等情况，这时需要在不中断业务的情况下将虚拟机迁移出去。虚拟机热迁移是将源物理机上指定的处于运行状态的虚拟机迁移到另一台物理机上，并保证在迁移过程中虚拟机业务不中断、用户不感知。

根据虚拟机数据存储在本地存储还是共享存储池上的不同，DCS 支持的虚拟机热迁移方式可以分为共享存储热迁移和非共享存储热迁移两种情况。如图 3-29 所示，共享存储热迁移是将虚拟机的 CPU、内存状态迁移到目的端的虚拟机，而非共享存储热迁移是在迁移虚拟机的 CPU、内存状态的基础上，将虚拟机镜像的全部数据迁移到目的端的镜像文件。

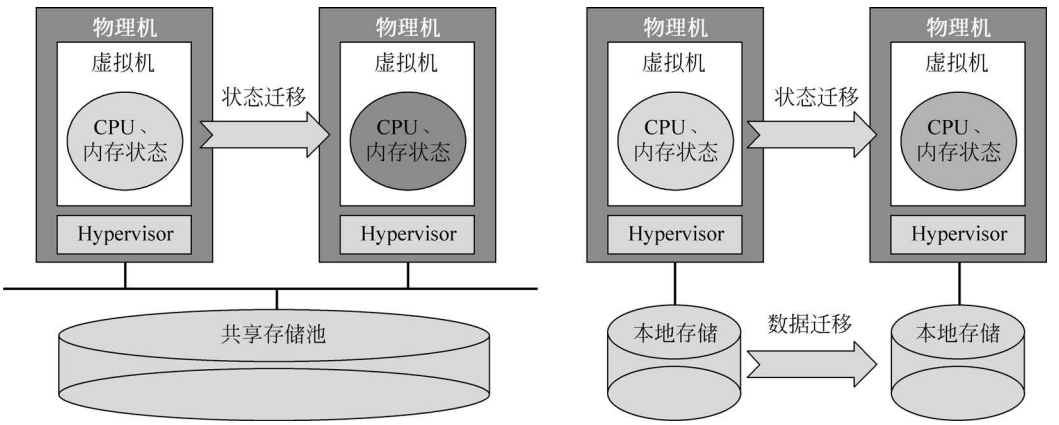


图 3-29 虚拟机热迁移的两种场景

虚拟机热迁移的大致流程如下（以共享存储为例）。

（1）对虚拟机的状态进行校验，虚拟机状态为运行中才能进行热迁移。

（2）选择目标节点并预占资源，启动一台暂停状态的虚拟机。

（3）源主机将 CPU、内存状态复制到目标主机的虚拟机，数据复制达到一定条件后触发事件，源主机暂停虚拟机。

（4）虚拟机暂停时，将未复制的数据一次性复制到目标主机的虚拟机。

（5）停止源主机的虚拟机，恢复目标主机的虚拟机。

在进行虚拟机热迁移时，也需要注意以下问题。

(1) 带宽在热迁移过程中至关重要,很大程度上决定了热迁移的速度。

(2) 由于热迁移过程中虚拟机是不停机的,所以会不断有业务程序产生新的数据,这部分数据被称为脏页数据。如果脏页数据产生的速度大于带宽,那么迁移将无法完成。

3.6.4　昇腾 NPU 虚拟化

在数据中心虚拟化场景下,虚拟机独占昇腾 NPU 资源可以获取最大的 NPU 推理和训练性能,但是很多业务并不需要单张卡的完整算力就能满足业务诉求。DCS 支持昇腾 NPU 虚拟化方案,在一个 NPU 物理设备下可以创建出多个 vNPU 设备,从而实现多个虚拟机共享同一张 NPU 卡的算力和资源,兼顾了性能和成本。

昇腾 NPU 虚拟化方案基于 Linux 内核支持的 MDEV 方案(见 3.5.2 节),如图 3-30 所示,其各组件实现如下。

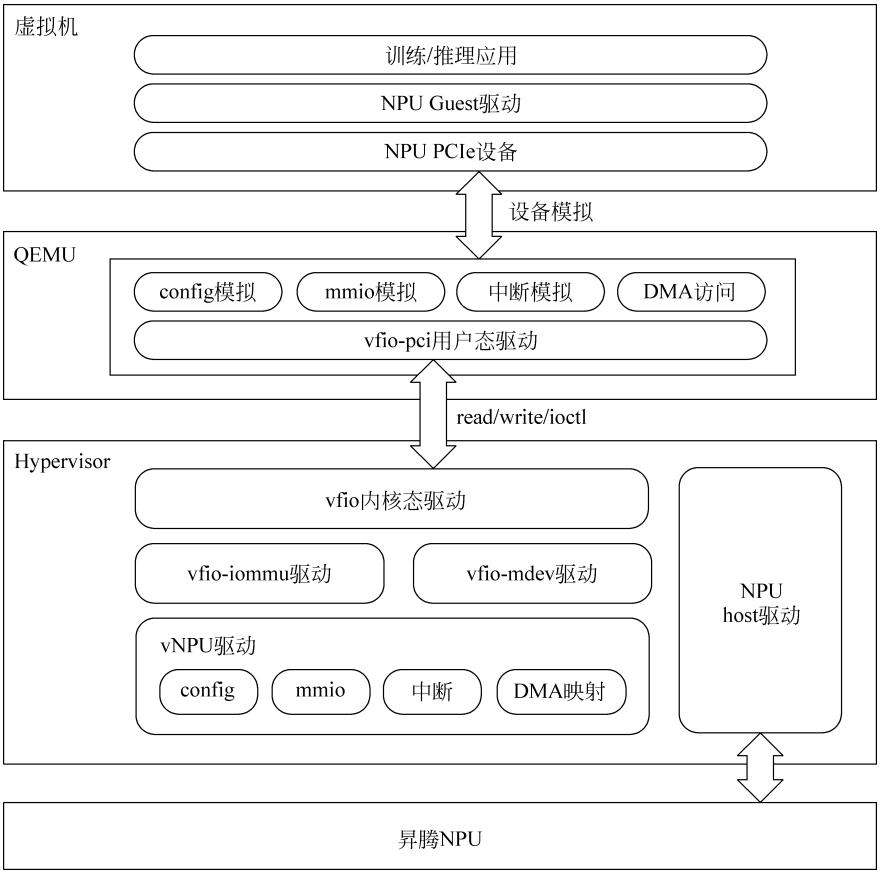


图 3-30　NPU 虚拟化

(1) GuestOS 内部的 NPU 驱动,需要适配 MDEV 虚拟化方案,对接 MDEV 的接口。

(2) QEMU 组件对 NPU 进行设备模拟,需要模拟 vNPU 设备的 config(配置空间)、mmio、中断以及 DMA 访问。

（3）Hypervisor 层的 vNPU 驱动，需要支持 MDEV 设备模拟，包括 config、mmio、中断、DMA 地址的映射。

NPU 虚拟化的流程可以分为以下步骤。

（1）设备创建：安装了 NPU host 驱动和 vNPU 驱动后，会向 MDEV 框架注册 NPU 设备。用户通过向 MDEV sysfs(System File System，是一种虚拟文件系统，提供了一种在 Linux 系统中管理设备和内核参数的机制)写入 uuid 来创建 vNPU 设备。

（2）设备模拟：将创建好的 vNPU 设备传给 QEMU，QEMU 通过 vfio 直通方式在虚拟机中模拟出一个正常的 NPU PCIe 设备。

（3）设备使用：在虚拟机内部安装 NPU Guest 驱动，该驱动会通过 MDEV 接口执行 config、mmio、中断和 DMA 访问等操作，从而在虚拟机内正常运行训练/推理应用。

3.7 小结

本章通过简要阐述 CPU 虚拟化、内存虚拟化、I/O 虚拟化以及新兴的 GPU 虚拟化技术，剖析了虚拟化技术如何提升物理资源的利用率，并实现多个操作系统和应用的并行高效运行。此外，本章还结合华为 DCS 解决方案，对计算虚拟化在实施案例中的应用进行了更加直观的展示。本章内容有助于读者对计算虚拟化的基本功能及面临的挑战有更全面的了解，思考如何运用硬件辅助虚拟化技术解决这些问题。