第5章

鲲鹏虚拟化

前几章介绍了 Intel x86 架构中硬件辅助虚拟化的设计与实现。本章将目光聚焦于另一种主流架构——ARMv8 的虚拟化硬件支持。本章同样涵盖虚拟化的四个基本要素：CPU、内存、I/O 和时钟。5.1 节简要介绍 ARM 虚拟化架构，5.2 和 5.3 节分别介绍 ARMv8 CPU 虚拟化以及中断虚拟化，5.4 节简要介绍 ARMv8 架构下的内存虚拟化，5.5 节主要介绍 ARMv8 架构下的 I/O 虚拟化，5.6 节简要介绍 ARMv8 架构下的时钟虚拟化。

5.1 鲲鹏虚拟化框架

鲲鹏系列处理器是基于 ARMv8 架构设计的面向服务器市场的处理器，下面先简要介绍 ARMv8 对于虚拟化的支持。

5.1.1 鲲鹏虚拟化简介

由于 ARMv8 需要对 ARMv7 进行兼容和升级，所以先介绍 ARMv7 的特权级和工作模式，并和 ARMv8 以及 x86 进行对比。

ARMv7 是 ARM 的 A 系列处理器对应的 32 位 ARM 架构。该架构提供了 7 种工作模式，分别是用户(USR)模式、系统(SYS)模式、一般中断(IRQ)模式、快速中断(FIQ)模式、管理(SVC)模式、中止(ABT)模式和未定义指令终止(UND)模式。七种模式中，用户模式称为非特权模式，其他模式称作特权模式，有权访问所有的系统资源。特权模式中除系统模式外的其余五种模式又称为异常模式。在这七种模式中用户模式的特权级最低，用户的应用程序运行在此模式下，无权访问硬件资源，只能通过模式切换、软中断或者异常等方式使 CPU 进入特权模式对硬件资源进行间接访问。ARMv8 架构在 ARMv7 的基础上进行了升级，支持 64 位架构并增加了异常级。

类似于 x86 架构的 Ring0～Ring3 特权级，ARMv8 提供了 EL0～EL3 四个异常级(Exception Level)。不同的是，在 ARM 中 EL0 的权限级别最低，而 EL3 权限级别最高。相比于 ARMv7 只有非特权模式和特权模式的两种特权级别，ARMv8 的四个异常级实际上是对特权级的一种扩充，用户程序运行在 EL0 异常级，操作系统则运行在 EL1 异常级。为了兼容 v7 运行模式，v8 将 v7 中的特权模式映射在 EL1 异常级。而更高的异常级 EL2 和 EL3 则分别运行 Hypervisor 和安全监视器(Secure Monitor)。ARMv8 整体异常级架构与对应的模式分布如图 5-1 所示。

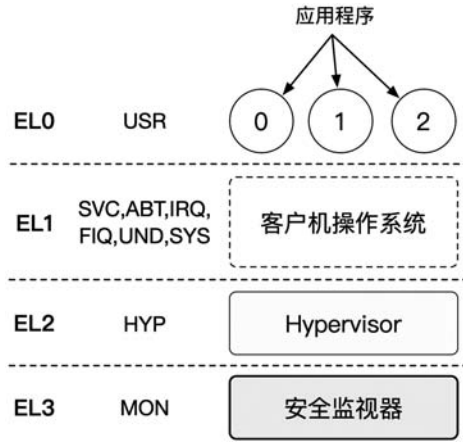


图 5-1　ARMv8 异常级与工作模式分布

ARMv8 还为各异常级增加了对应的寄存器(如 * _EL1、* _EL2 等),并将 Hypervisor 运行在比客户机操作系统更高一级的 EL2 异常级上,为 ARM 虚拟化提供了硬件支持。EL2 异常级的存在也实现了 x86 架构特权级"-1 级"的预想方案。

5.1.2　EL2 虚拟化框架

类似于 x86 的虚拟化框架模型,ARMv8 虚拟化也有两种类型:Type Ⅰ和 Type Ⅱ。Type Ⅰ类似于 Hypervisor 模型,Type Ⅱ类似于宿主机模型。

(1) **Type Ⅰ**:在这种模型中,Hypervisor 运行在 EL2 异常级,控制运行在 EL1 异常级中的虚拟机,对虚拟机进行隔离和必要的资源共享管理,如图 5-2 所示。在这种模式下,虚拟机要想获取全局资源就需要产生异常从而陷入位于 EL2 中的 Hypervisor 进行处理。这样虽然对虚拟机进行了有效的隔离,但是特权级切换时引入的上下文切换造成了虚拟机的性能损耗。ARM 为此设计了 VHE(Virtualization Host Extensions,虚拟化主机拓展)技术,也就是 Type Ⅱ模型。

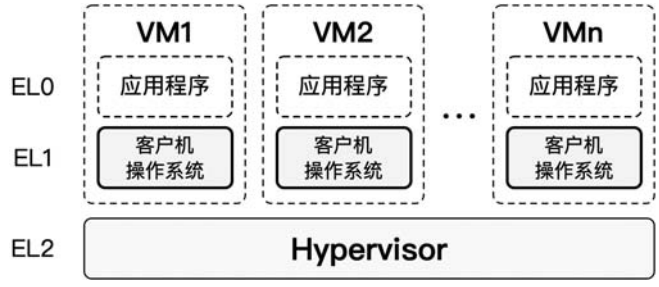


图 5-2　ARMv8 的 Type Ⅰ虚拟化模型

(2) **Type Ⅱ**:与 Type Ⅰ不同的是,在 Type Ⅱ模型中 EL2 异常级上运行的是宿主机操作系统,Hypervisor 则依赖于宿主机操作机系统,利用宿主机操作系统运行环境来管理虚拟

机。当发生异常时，异常级会直接由 EL0 切换到 EL2，略去了到 EL1 的切换步骤，节省了上下文切换的开销，提升了虚拟化性能，详见 5.2.3 小节 VHE 虚拟化。

5.2　鲲鹏 CPU 虚拟化

第 2 章提出了 CPU 虚拟化所面临的三个挑战以及 Intel VT-x 相应的解决方案，在某种程度上，这也是 ARMv8 硬件辅助虚拟化需要解决的问题。本节将介绍 ARMv8 的解决方案以及在设计上与 Intel VT-x 的不同之处。

5.2.1　CPU 虚拟化

如前所述，CPU 虚拟化本质上通过创建 QEMU 线程来创建 vCPU。那么 vCPU 是如何运行以达到和真正的 CPU 一样的效果呢？鲲鹏处理器使用 VHE 扩展，将 KVM 和 Linux 内核代码直接运行在 EL2，管理控制运行在 EL0 中的 QEMU 线程，对外呈现出独立的 CPU 特性，其具体实现如下。

首先是 Hypervisor 初始化，设置一些与虚拟化相关的寄存器并开启虚拟化模式，然后开始运行 vCPU。vCPU 的主要任务是模拟真实的 CPU，在鲲鹏虚拟化中采取的是 QEMU 和 KVM 相结合的方式。为了隔离多个虚拟机的运行环境，vCPU 执行的指令将受到 Hypervisor 的管理与限制。但是如果每一条指令都要经过 Hypervisor 处理必然会造成性能损耗。因此将指令分为敏感指令和非敏感指令。非敏感指令交由运行在 EL0 级别的 QEMU 线程直接执行，敏感指令则交由 Hypervisor 进行处理。敏感指令的识别交由硬件完成，CPU 自动识别指令是否是敏感指令。对于普通的指令（用户 ISA），CPU 可以直接运行，而对于敏感指令的执行（系统 ISA），vCPU 处理较为复杂，下面举例说明。当执行到一条敏感指令时，CPU 会自动识别这条指令，然后触发指令陷入。由于鲲鹏处理器支持 VHE 技术，所以敏感指令将会直接陷入 EL2 级别的 Hypervisor 中。在陷入的过程中，CPU 需要依次完成如下工作。

（1）**保存上下文**。在虚拟机退出之前要保存上下文，以便执行完敏感指令之后继续运行虚拟机。

（2）**执行敏感指令**。为了保证虚拟机之间的隔离性，敏感指令不能在 EL2 级直接执行，而是通过模拟的方式执行。以关机、休眠等指令为例，Hypervisor 会通过模拟的方式使虚拟机关机或者休眠，而不会影响主机。

（3）**恢复上下文**。将第一步中保存的上下文恢复到相应的寄存器中，具体过程不再赘述。

经过上述三个步骤，vCPU 可以模拟敏感指令的执行。然而，CPU 虚拟化技术不仅仅是提供指令的执行单元这么简单，还可以使 vCPU 数目超过物理 CPU 的数量。Hypervisor 通过调度不同的 vCPU 来使用物理 CPU 完成对 CPU 的虚拟，类似于操作系统调度任务时采用的分时复用思路。vCPU 的数量可以超出真实 CPU 的数量，因为 vCPU 就是 QEMU

中的一个线程。但是如果 vCPU 的数量过多，反而会降低整个虚拟机的性能，因为 Hypervisor 在调度 vCPU 时，上下文切换会带来一定的时间损耗。

5.2.2 EL2 异常级

继 Intel 和 AMD 相继推出硬件辅助虚拟化拓展后，ARM 于 2012 年推出了自己的硬件虚拟化拓展。与 Intel VT－x 类似，ARMv8 硬件辅助的 CPU 虚拟化旨在解决“虚拟化漏洞”问题，使所有的敏感指令都能触发虚拟机下陷，从而使 Hypervisor 能够截获并模拟敏感非特权指令的执行。Intel VT-x 通过引入根模式与非根模式两种操作模式，并改变非根模式下敏感非特权指令的语义使其触发 VM-Exit，解决了“虚拟化漏洞”问题。ARMv8 则引入了 EL2 来解决上述问题，ARMv8 异常级架构如图 5-3 所示。

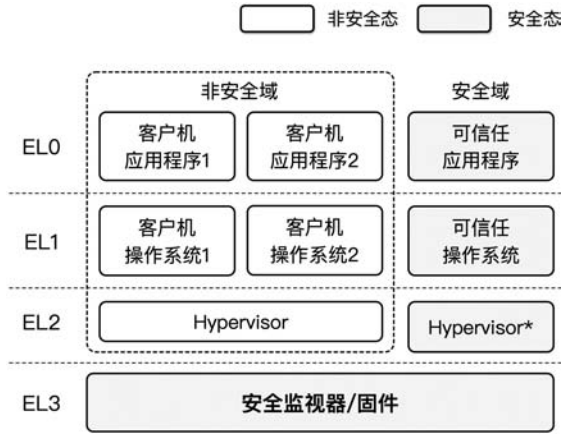


图 5-3 ARMv8 异常级架构

为了保障系统的安全，ARMv8 引入了安全状态(Security State)这一概念。处理器可以处于安全态(Secure State)或非安全态(Non-secure State)，二者都有 EL0～EL2[①] 异常级以及独立的物理地址空间，安全状态与异常级别共同组成了处理器的当前状态。对于安全态，它拥有完整的物理资源控制权限，可以访问两种状态下的地址空间以及系统寄存器，而非安全态只能访问自己的地址空间以及部分系统寄存器。通常将可信的操作系统和应用运行在安全态，普通的操作系统和应用，如 Linux 和 Android 操作系统，则运行在非安全态，从而避免了不可信应用带来的安全隐患。异常级则类似于 x86 中特权级的概念，其中 EL0 异常级最低，EL3[②] 异常级最高。当 ARMv8 未开启虚拟化拓展时，应用程序运行在 EL0，操作系统则运行在 EL1，有权访问所有的硬件资源。而当开启了虚拟化拓展后，Hypervisor 运行在 EL2，客户机操作系统运行在 EL1，客户机应用程序运行在 EL0。ARMv8 提供了 HCR_EL2 寄存器，用于控制虚拟机行为，类似于 x86 VMCS 中的 VM-Execution 控制域。

① ARMv8.4 引入了安全态 EL2(Secure EL2)，将 SCR_EL3.EEL2 置 1 开启安全态 EL2。

② 认为 EL3 始终处于安全态，安全态的切换由 EL3 完成。

以 HCR_EL2.TWI 为例，在 EL0 和 EL1 中执行 WFI 指令会导致其陷入 EL2 中。在物理环境下，WFI 指令会使当前 CPU 进入低功耗状态；而在虚拟环境下执行 WFI 指令将会令 Hypervisor 调度另一个 vCPU 运行。尽管硬件辅助虚拟化已大大减少了虚拟化的开销，但是频繁的虚拟机下陷引起的异常级切换仍会引入巨大的虚拟化开销。ARMv8 对操作系统频繁访问的寄存器进行了进一步优化，如 MIDR_EL1 和 MPIDR_EL1 寄存器。MIDR_EL1 保存着处理器的类型，MPIDR_EL1 则保存着处理器亲和性相关的信息。对于这两个寄存器，Hypervisor 更希望客户机操作系统能够直接读取到虚拟机中相应寄存器的值，而无须陷入 Hypervisor 中。故 ARMv8 提供了 VPIDR_EL2 和 VMPIDR_EL2 寄存器，Hypervisor 在运行虚拟机之前配置好这两个寄存器的值，当虚拟机读取 MIDR_EL1/MPIDR_EL1 时，会自动返回 VPIDR_EL2/VMPIDR_EL2 的值。此外，不同于 Intel VT-x 使用 VMCS 保存虚拟机上下文，ARMv8 直接为 EL1 和 EL2 提供了两套系统寄存器，这样虚拟机下陷时便无须保存虚拟机寄存器状态，降低了上下文切换的开销。而运行在 EL2 中的 Hypervisor 可以直接读写 EL0/EL1 中的寄存器。

即便如此，当运行在 EL2 的 Hypervisor 需要操作系统内核服务时，仍需切换到 EL1 异常级进行操作，这样还是会有大量的异常级切换损耗。为此在 ARMv8.1 中新增了虚拟化主机扩展 VHE。

5.2.3　VHE

硬件层面上，VHE 主要增加了以下几个部分。

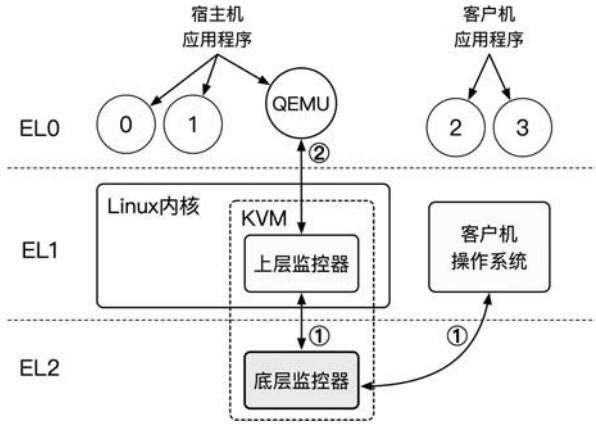
(1) 在 EL2 级别的 Hypervisor 配置寄存器中增加了用于指示是否开启 VHE 的控制位 E2H。

(2) 在 EL2 级别新增了 TTBR1_EL2、CONTEXTIDT_EL2 寄存器供宿主机操作系统使用。

(3) 增加了新的虚拟计时器。

当宿主机内核启动后，首先会调用 stext，stext 调用 el2_setup 根据内核是否配置了 CONFIG_ARM64_VHE 来决定是否开启了 VHE。开启 VHE 后，宿主机操作系统会直接运行在 EL2。当产生虚拟机下陷时，由于宿主机运行在 EL0(宿主机应用程序)和 EL2(宿主机操作系统)两个异常级，所以会直接从虚拟机运行的 EL0 异常级直接切换为 EL2 异常级。

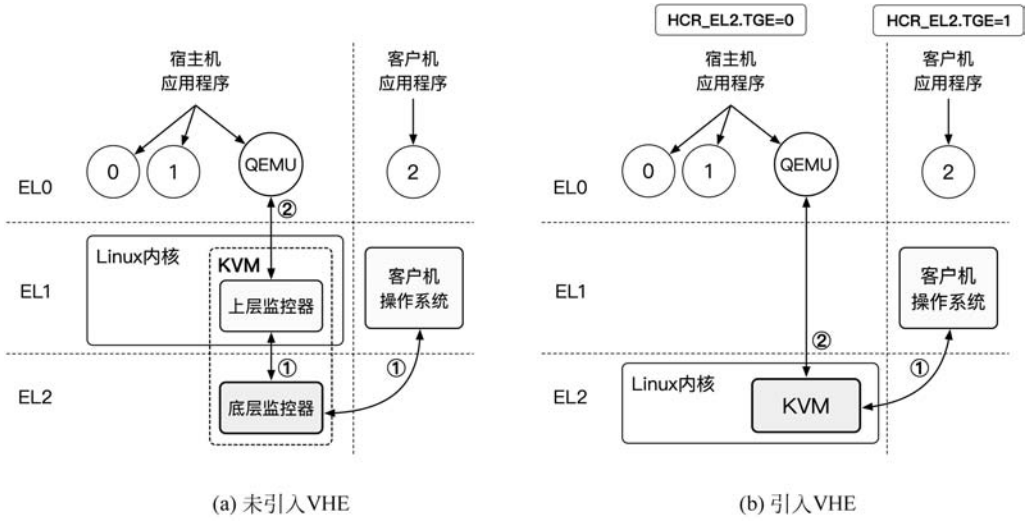
图 5-4 展现了 ARMv8 虚拟化架构，Hypervisor 运行在 EL2 异常级，虚拟机则运行在 EL0/EL1 异常级，但是该架构只支持 Type Ⅰ类型的虚拟机，不支持 Type Ⅱ类型的虚拟机。这是因为 Linux 等操作系统开发时假定其运行在 EL1 异常级，部分 EL1 的寄存器在 EL2 中并不存在；而在 Type Ⅱ类型的虚拟机中，Hypervisor 很大程度上依赖于宿主机操作系统提供的接口，因此就出现了宿主机操作系统运行在 EL1，而 Hypervisor 运行在 EL2，二者异常级不一致的问题。以 KVM 为例，为了解决上述问题，系统开发人员提出将 KVM 划分为上层监控器(Highvisor)和底层监控器(Lowvisor)两部分，其中底层监控器运行在 EL2，上层监控器运行在 EL1，架构如图 5-4 所示。



注：①超级调用(hypercall)；②系统调用(syscall)。

图 5-4　ARM KVM 架构图

当发生虚拟机下陷时，首先会进入 EL2 中，底层监控器进行必要的处理，当它需要使用 Linux 内核的功能时，则切换到 EL1 中的上层监控器进行处理。通常底层监控器相对精简，只进行必需的处理，将大部分工作留给上层监控器处理。分离模式虚拟化解决了在 ARMv8 上运行 Type Ⅱ类型虚拟机的问题，但是这种分层模式造成了大量的上下文切换，严重影响虚拟化性能，于是 VHE 应运而生，它允许宿主机操作系统运行在 EL2，从硬件层面解决了上述问题。引入 VHE 前后 Type Ⅱ类型虚拟机运行如图 5-5 所示。VHE 由 HCR_EL2. E2H 和 HCR_EL2. TGE 控制，E2H 用于使能 VHE，而 TGE 用于在使能 VHE 时区分虚拟机应用程序和物理机应用程序。



注：①超级调用(hypercall)；②系统调用(syscall)。

图 5-5　引入 VHE 前后的虚拟化架构

在 ARMv8 的虚拟化中，虚拟化组件有两个：一个是运行在 EL0 异常级的 QEMU，另一个是运行在 EL2 异常级的 KVM。类似于 x86 下的 KVM 架构，ARM 的虚拟化也是通过 QEMU 线程来模拟 vCPU，通过 ioctl 命令在 QEMU 和 KVM 之间交互。在 QEMU 需要获取全局资源的时候，执行 VM-Exit 操作切换到 EL2 异常级进行全局资源的处理。同时 ARM 在虚拟化设计中也有自己独特之处。

在 VHE 中，由于宿主机操作系统需要运行在 EL2 异常级，所以就需要访问 EL2 的寄存器。但是由于现有的操作系统都是运行在 EL1 异常级上面的，它们会默认访问 EL1 的寄存器。为了能够不加修改地在 EL2 级运行现有的操作系统内核，就需要对宿主机操作系统进行寄存器重定向操作。具体的方法为：根据是否开启 VHE 的标志位 E2H，来判断是否需要对寄存器进行重定向。当 E2H 为 1 时，运行在 EL2 异常级的指令进行寄存器重定向，而当 E2H 为 0 时则不用重定向。

但是重定向又引入了一个新的问题：如果运行在 EL2 异常级的 Hypervisor 确实需要访问 EL1 的寄存器，那么重定向就会将其定向到 EL2 的寄存器上。为此 ARM 架构引入了新的别名机制：以_EL12 或者_EL02 结尾。当访问这些别名时，就可以正常访问 EL1 的寄存器。

5.3　鲲鹏中断虚拟化

由于鲲鹏服务器搭载 ARM 架构的芯片，所以其使用的中断控制器为 ARM 架构采用的 GIC(Generic Interrupt Controller，通用中断控制器)。GIC 发展至今已历经四代，本节主要侧重于最新的 GICv3/GICv4 架构，将不会深入讲述 GICv1/GICv2 架构。

5.3.1　GICv1

GICv1 是 ARM 最早推出的中断控制器，现在已经弃用，其架构如图 5-6 所示。GICv1 最多支持 8 个 PE(Processing Element，处理器单元)和 1020 个中断源。中断控制器的加入使得处理器可以及时地响应外部设备发送的请求。当外围设备发送中断请求时，中断控制器可以及时捕获并在经过仲裁后将中断信号发送给 CPU，然后等待 CPU 对于中断的处理结果。遗憾的是 GICv1 并不支持中断虚拟化。

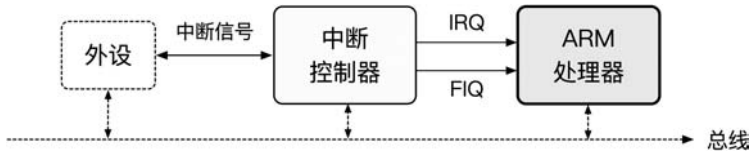


图 5-6　GICv1 架构

5.3.2　GICv2

GICv2 增加了对中断虚拟化的支持，但是仍只支持 8 个 PE，其架构如图 5-7 所示。

上面的中断控制器主要分为以下几个部分：中断分发器（Distributor）、CPU 接口（CPU Interface）和 CPU 虚拟接口（CPU Virtual Interface），其中 CPU 虚拟接口主要用于中断的虚拟化。各部分的主要功能如下。

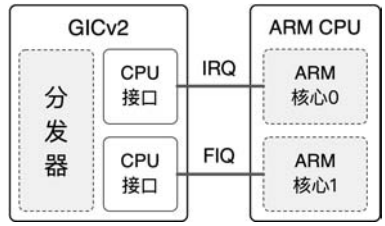


图 5-7 GICv2 架构

（1）分发器：主要用于处理中断的优先级，并将中断分发给对应的 CPU 接口。

（2）CPU 接口：主要用来处理中断相关事务，如中断优先级屏蔽、中断抢占以及与 CPU 之间的通信等。每个 CPU 核都有一个 CPU 接口。

（3）CPU 虚拟接口：主要用在虚拟化环境，是虚拟 CPU 的 CPU 接口。在虚拟化场景中，需要将 HCR_EL2.IMO 设置为 1，此时所有的中断信号都将会陷入 Hypervisor 中，并由 Hypervisor 判断是否将该中断信号插入 vCPU 中。

GICv2 共支持 1020 个中断源，根据中断的编号将中断分为以下三类。

（1）SGI（Software Generated Interrupt，软件生成中断）：由编号为 0 到编号为 15 的中断源组成。这种中断由 CPU 直接写对应的寄存器触发，而非硬件触发，所以叫作软件产生的中断。这种中断主要用于 ARM 核间通信。

（2）PPI（Private Peripheral Interrupt，私有设备中断）：由编号为 16 到编号为 31 的中断源组成。该中断源为 CPU 私有的中断源，类似于 x86 中的 LAPIC。

（3）SPI（Shared Peripheral Interrupt，共享设备终端）：由编号为 32 到编号为 1019 的中断源组成。该中断源是所有 CPU 共享的中断源，类似于 x86 中的 IOAPIC。而编号为 1020 到编号为 1023 的中断源预留做其他用途。

5.3.3 GICv3/GICv4

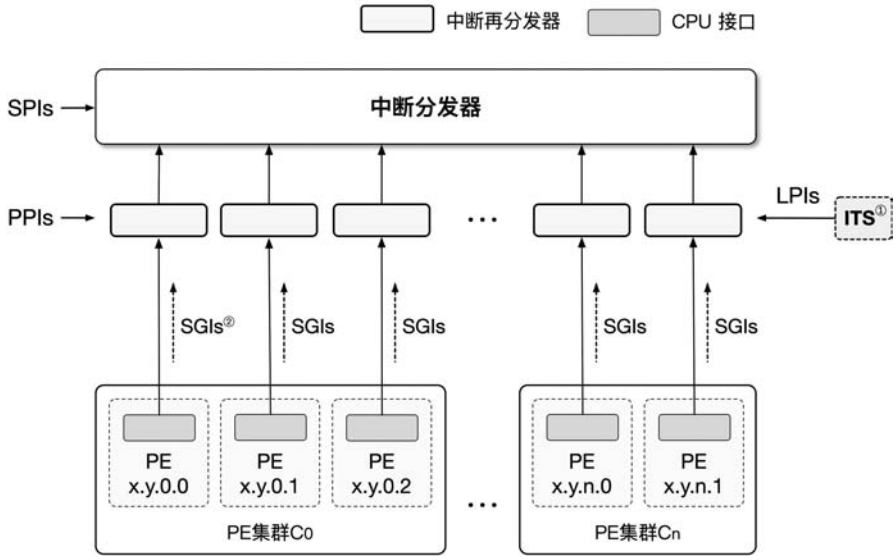
相较于 GICv2，GICv3 增加了许多新功能，而 GICv4 相较于 GICv3 则变化不大。在后文中，未经特殊说明，GIC 都是指 GICv3 架构。图 5-8 中不同颜色的矩形表示 GIC 架构中的组件，箭头则表示中断传递的流程。GICv3 架构主要包括四个组件：中断分发器（Distributor）、中断再分发器（Redistributor）、CPU 接口（CPU Interface）和 ITS。在正式介绍这四个组件的功能之前，需要先了解 GIC 中的一些基本概念和机制。

1. 中断类型

除了 GICv2 定义的三种中断类型外，GICv3 还引入了一种新的中断类型 LPI（Locality－specific Peripheral Interrupt，特殊设备中断）。LPI 是 GICv3 引入的一种新的基于消息的中断类型，可以兼容 PCIe 总线的 MSI 和 MSI－X 机制。

2. 中断 ID

GIC 为每个中断指定一个 INTID（Interrupt ID，中断 ID），类似于 x86 的中断向量号。但是不同于 x86 中段向量号暗含中断优先级，GIC 显式地为每个中断都指定了一个中断优



注：①可能存在 0 个或多个 ITS；②SGIs 由 PE 产生，由中断分发器路由。

图 5-8　GICv3 中断架构

先级号。通常中断优先级号越小，优先级越高。GIC 允许高优先级中断抢占低优先级中断。GIC 将中断优先级号分为两部分：优先级组号（Group Priority）和次优先级号（Sub-Priority），中断抢占需要满足以下两个条件。

（1）阻塞中断的优先级组号小于 CPU 当前的运行优先级组号，即当前正在被处理的最高优先级中断的优先级组号。

（2）阻塞中断的优先级组号小于当前 CPU 的屏蔽优先级组号（Priority Mask）。

SGI/PPI 类型中断的优先级号保存在中断再分发器的相关寄存器中，SPI 类型中断的优先级号保存在中断分发器的相关寄存器中，LPI 类型中断的优先级号则保存在内存中的 LPI 配置表中。

3. 中断分组

GIC 还引入了中断分组（Interrupt Grouping）机制使得特定的中断只能被特定的异常级处理，从而保障系统安全。为了兼容如图 5-1 所示的异常级架构，GICv3 引入了中断分组机制，它将物理中断分为三组。

（1）组 0（Group 0）：ARMv8 期望这些中断在 EL3 处理。

（2）安全组 1（Secure Group 1）：ARMv8 期望这些中断在安全态 EL1（Secure EL1）中处理。

（3）非安全组 1（Non-secure Group 1）：在虚拟化环境下，ARMv8 期望这些中断在非安全态（Non-secure EL2）中处理；在非虚拟化环境下，ARMv8 期望这些中断在非安全态 EL1（Non－secure EL1）中处理。

SGI/PPI 类型中断的分组保存在中断再分发器的相关寄存器中，SPI 类型中断的分组保存在中断分发器的相关寄存器中，而 LPI 类型中断一定属于非安全组 1。

4. ITS

ITS 是 GICv3 新引入的组件，它输出 LPI 类型的物理中断。外设只需要提供设备号(Device ID)和事件号(Event ID)就能触发一个 LPI 中断，其中事件号需要写入 ITS 中的 GITS_TRANSLATER 寄存器，而设备号的传输是由架构具体实现所定义的。ITS 在内存中维护了四种类型的表：DT(Device Table，设备表)、ITT(Interrupt Translation Table，中断翻译表)、CT(Collection Table，集合表)和 vPE Table(Virtual PE Table，虚拟 PE 表)。其中 vPE Table 是 GICv4 添加的支持，使得 ITS 还可以输出 LPI 类型的虚拟中断，且无需 Hypervisor 介入便可以将虚拟 LPI 中断注入 vCPU。各表之间的关联如图 5-9 所示。

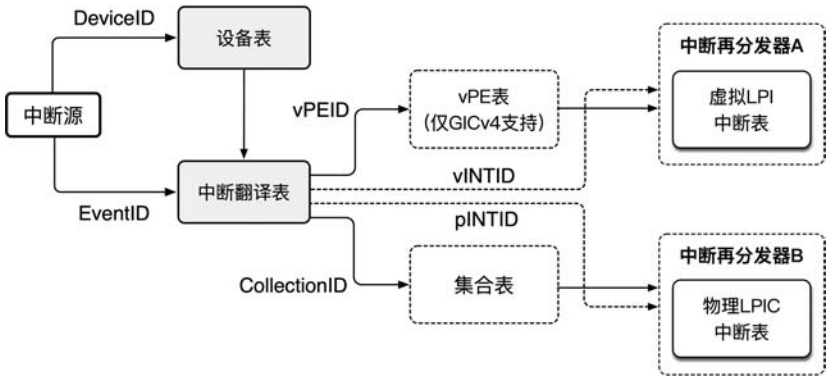


图 5-9 ITS 表

各类型表的主要功能如下。

(1) 设备表：维护了设备号和中断翻译表基地址的映射，ITS 为每个设备单独维护一个中断翻译表。

(2) 中断翻译表：中断翻译表可以维护两种类型的映射。对于物理环境而言，中断翻译表通过事件号得到该事件对应的物理 LPI 中断的 INTID 和 ICID(Interrupt Collection ID，中断集合 ID)，然后 ITS 用 ICID 索引集合表得知该中断的目的中断再分发器。对于虚拟环境而言，中断翻译表通过事件号得到该事件对应的虚拟 LPI 中断的 INTID 和 vPEID，然后 ITS 用 vPEID 索引虚拟 PE 表得知该虚拟中断的目的 vCPU，将该虚拟中断发送至 vCPU 所在物理 CPU 对应的中断再分发器。若当前 vCPU 没有运行，ITS 将给物理 CPU 发送一个门铃中断(Doorbell Interrupt)，使其调度 vCPU 运行。

(3) 集合表：如上所述，维护 ICID 和中断再分发器的映射。

(4) 虚拟 PE 表：如上所述，维护 vPEID 和中断再分发器的映射。

5. GIC 组件

前面提到，GIC 中断架构主要包括四个部分：中断分发器、中断再分发器、CPU 接口和

ITS。其中中断分发器、中断再分发器和 ITS 实现在 GIC 内部，故 CPU 通过 MMIO 的方式访问其内部寄存器。而 CPU 接口则位于 CPU 核内部，可以直接通过系统寄存器访问。中断分发器和 ITS 由所有 CPU 核共享，中断再分发器则与 CPU 核一一对应，每一个 CPU 接口都有一个中断再分发器与其相连，各部分具体功能如下。

(1) 中断分发器：主要维护 SPI 类型中断的相关信息，如中断优先级、中断分组、中断路由信息和中断状态等。此外，SGI 类型中断也通过中断分发器路由。GICv3 引入了亲和性路由(Affinity Routing)机制，使得 SPI 路由至指定 CPU 或指定 CPU 集合中的某一个，而 SGI 则会被路由至指定 CPU 集合中的每一个 CPU。

(2) 中断再分发器：维护 PPI/SGI 类型中断的相关信息，如中断状态、中断优先级、中断触发方式等。

(3) ITS：ITS 是 GICv3 新引入的组件，它输出 LPI 类型的中断。外设只需要向 ITS 的 GITS_TRANSLATER 寄存器写入一个事件号(EventID)就能触发一个 LPI 中断。而在 GICv4 架构中，ITS 还提供了 Virtual LPI 直接注入机制，类似于 APICv 提供的分布-中断机制，无需 Hypervisor 参与便能直接注入虚拟中断。

(4) CPU 接口：主要负责中断优先级检测、优先级仲裁(选择优先级最高的中断处理)和中断应答等。

6. GIC 中断处理

在 GIC 中断架构中，中断存在以下四种状态。

(1) 非活跃态(Inactive)：当前没有待处理或正在处理的中断。

(2) 阻塞态(Pending)：当前中断正在等待 CPU 处理。

(3) 活跃态(Active)：当前中断已经被 CPU 响应，该中断正在被 CPU 处理。

(4) 活跃阻塞态(Active and Pending)：当前中断正在被处理时，又收到一个相同 INTID 的中断。

四种中断状态在中断处理过程中的变化如下所述。

(1) **中断产生**：外部设备或系统程序通过中断连接线或写入 GIC 相关寄存器触发中断，此时中断从非活跃态变为阻塞态。

(2) **中断分发**：GIC 通过前述寄存器或内存控制结构确定该中断的优先级、中断分组等信息，并通过亲和性路由机制将中断发送给目标 CPU 接口。

(3) **中断交付**：CPU 接口将中断递交给 CPU。

(4) **中断激活**：CPU 应答该中断，表明该中断正在被处理，该中断由阻塞态变为活跃态。在此过程中如果有相同 INTID 的中断到达，则中断变为活跃阻塞态。

(5) **运行优先级降低**：CPU 处理完该中断后，首先修改当前的运行优先级，使得低优先级中断可以被响应，此时中断仍处于活跃态。

(6) **中断无效**：将当前中断状态置为非活跃态，使得后续处于阻塞态的同 INTID 中断能够被处理。

5.3.4 GICv3/GICv4 中断虚拟化

GIC 为中断虚拟化提供了以下硬件支持：虚拟 CPU 接口寄存器直接访问、虚拟中断注入以及虚拟 LPI 中断直接注入。

1. 虚拟 CPU 接口寄存器直接访问

在早期 GIC 架构中，CPU 接口也位于 GIC 内部，同中断分发器等组件一样，CPU 通过 MMIO 的方式访问其内部寄存器。而在虚拟环境中，为了避免虚拟机直接写入 CPU 接口中的寄存器，Hypervisor 通常会将接口寄存器映射区域设置为不可访问，从而截获所有访问并陷入 Hypervisor 中进行相应的模拟。这一处理方式同前述早期 x86 中断控制器的访问类似。而 GICv3 将 CPU 接口移入 CPU 内部并提供相关的系统寄存器（ICC_*）供 CPU 使用，大大提升了中断响应速度。而为了应对虚拟化环境，GICv3 还为这些系统寄存器提供了相应的虚拟 CPU 接口寄存器（ICV_*），当 Hypervisor 将 HCR_EL2.IMO 和 HCR_EL2.FMO 设置为 1 时，运行在 EL1 中的虚拟机操作系统对 CPU 接口系统寄存器（ICC_*）的访问将被重定向到相应的 ICV_* 寄存器，从而避免了中断处理过程中访问 CPU 接口寄存器造成的虚拟机下陷。

2. 虚拟中断注入

GICv3 可以配置使得所有的物理中断路由到 EL2，此时 Hypervisor 对该中断进行检查，若该物理中断的目标为 Hypervisor，则 Hypervisor 按照前述流程处理该物理中断；若该中断目标为 vCPU，则 Hypervisor 会向 vCPU 中注入一个虚拟中断。GICv3 提供了寄存器（ICH_LR<n>_EL2）保存虚拟中断的 INTID、中断优先级、中断状态以及相关联的物理中断 INTID 等。当 vCPU 恢复运行时，硬件将根据这些寄存器中的信息向 vCPU 注入一个虚拟中断，并调用相应的中断处理函数。

3. 虚拟 LPI 中断直接注入

GICv4 引入了虚拟 LPI 中断注入机制，无需 Hypervisor 参与便可以将虚拟 LPI 中断注入 vCPU，这是通过前述 GIC 的 ITS 组件完成的。ITS 通过查询设备表、中断翻译表和虚拟 PE 表得到虚拟中断目的 vCPU 所处的物理 CPU 所对应的中断再分发器，将虚拟 LPI 中断信息插入位于内存中的虚拟 LPI 配置表（Virtual LPI Configuration Table）和虚拟 LPI 状态表（Virtual LPI Pending Table）中。然后 vCPU 会将表中记录的虚拟 LPI 中断与前述 ICH_LR<N>_EL2 寄存器中记录的虚拟中断进行比较，选择最高优先级的中断进行处理。

5.4 鲲鹏内存虚拟化

ARM 架构借鉴了 Intel 系列架构演进过程的“前车之鉴”，在 ARMv7 架构中就包含了对双层页表的支持，保证了内存虚拟化的性能；而 Intel 在较晚的虚拟化版本中才支持扩展页表，在此之前只能使用软件实现的影子页表，性能较差。ARM 架构也为内存管理增加了

更多的寄存器，用于保存多类页表的基地址。Intel 系列仅有一个 CR3，并使用 VMCS 结构保存 EPT 页表基地址；而 ARM 架构为操作系统的用户态和内核态各准备了一个寄存器，用于保存页表基地址，增强了用户态空间和内核态空间的隔离性。下面从与 Intel 架构类似的地址翻译概念讲起，并逐步介绍 ARM 中出现的新概念。

5.4.1　VMSAv8-64 架构概述

在 ARM 处理器中，VMSA（Virtual Memory System Architecture，虚拟内存系统架构）给在 ARMv8 处理器的 AArch64 模式下运行的 PE 提供了虚拟内存管理功能。具体而言，VMSAv8-64 为 PE 提供了 MMU，当 PE 进行内存访问时，MMU 可以完成地址翻译、权限检查以及内存区域类型判断等功能。对于 PE 而言，它访问内存时使用的是 VA，MMU 可以：①将 VA 映射到 PA，用于访问物理内存系统中的资源。完成地址映射时，MMU 会使用保存着页表基地址的寄存器；②如果由于一些原因，无法将 VA 映射到 PA，则会引起异常（Exception），称为 MMU 异常（MMU Fault）。系统寄存器负责保存引起 MMU 异常的原因，供软件使用。

ARM 内存管理中的概念和 Intel 系列处理器中的大致相同。但如 5.1.2 节所述 ARM 异常级架构的介绍，ARM 中 PE 执行的异常级主要分为 EL0、EL1、EL2 等，此处仅考虑非安全状态下操作系统以及 Hypervisor 所运行的异常级。相应的，地址翻译系统相对于 Intel 系列有所改变。ARM 架构中提出了一个通用的概念，即翻译流程（Translation Regime），用来概括物理机环境和虚拟化环境下的内存翻译流程。ARM 中包括两类翻译流程：①单阶段的地址翻译，即 VA 翻译为 PA，是在物理机上运行的操作系统中发生的地址翻译流程；②两个连续阶段的地址翻译，即客户机虚拟地址翻译为客户机物理地址，进而翻译为宿主机物理地址，而 ARM 架构为了与前一种翻译流程的说法统一，将客户机虚拟地址依然称为 VA，客户机物理地址称为 IPA（Intermediate Physical Address，中间物理地址），宿主机物理地址称为 PA，阶段-1 的地址翻译将 VA 翻译为 IPA（Stage-1），阶段-2 的地址翻译将阶段-1 得到的 IPA 作为输入，翻译为 PA（Stage-2）。一个翻译流程定义了某异常级下地址翻译使用的页表基地址寄存器和控制寄存器，每个异常级各有其如下翻译流程。

（1）EL0&EL1：关闭 EL2 时，系统中不运行 Hypervisor，也不存在虚拟化的概念。EL0&EL1 异常级上的内存访问共用同一个单阶段的翻译流程，只进行单阶段的地址翻译，将 VA 翻译为 PA。EL1 运行着操作系统内核，EL0 运行着应用程序，均使用 VA 访问内存，需要翻译为 PA。此时在 TLB 中查找地址翻译缓存时，VMSA 系统需要根据 ASID 进行匹配，ASID 为每个进程标识了其独占的 TLB 表项，于是进程切换时无须清空 TLB，加快了地址翻译的速度。

（2）EL0&EL1：开启 EL2 时，运行在 EL1 的操作系统成为客户机操作系统，于是 EL0&EL1 异常级上的内存访问共用一个两阶段的翻译流程，即前文所述的 VA→IPA→PA。在 TLB 中查找地址翻译缓存时，VMSA 首先匹配相同的 VMID，筛选当前客户机独占的 TLB 缓存，在 vPE 切换时无须清空 TLB；然后匹配 ASID，筛选当前客户机进程的

TLB 缓存。

(3) 其他异常级(如 EL2、EL3)也可以使用 VA 访问内存。此时没有客户机的概念，故只需要单阶段的翻译流程将 VA 翻译为 PA。

如 5.1.2 节所述，与异常级正交的概念是安全状态，即每个 EL0、EL1、EL2 异常级都有安全态和非安全态，但与地址翻译的关系不大，故不做赘述，详细内容请参看 ARMv8 架构说明手册。ARM 架构在不开启虚拟化和开启虚拟化的情况下使用了类似的名词描述地址翻译过程，即阶段-1 和阶段-2，而 Intel 架构使用 EPT 指代阶段-2 使用的页表。ARM 提供了更多寄存器用于保存页表起始地址，这也和 Intel 架构形成了鲜明对比，下文将介绍。

5.4.2 地址空间与页表

在 ARM 虚拟化设计中，虚拟机应用要想访问物理内存必须经过两级转换：VM 维护一套地址转换表，Hypervisor 控制最终的转换结果。在第一级转换的过程中，虚拟机将其虚拟地址(VA)转化为虚拟机视角下的物理地址(IPA)，然后由 Hypervisor 最终控制转化为实际的物理内存地址。因此 Hypervisor 可以控制虚拟机访问特定大小的内存，并且指定被访问内存的空间位置。而其他的内存对于虚拟机来说都是不可见的，虚拟机无法访问。这种设计增强了虚拟机之间的隔离，保证了虚拟机的安全性。

IPA 作为中间物理地址的时候，不仅存在内存区，它还包含了外围设备区域。虚拟机可以通过 IPA 的外围设备区域来访问虚拟机可见设备。而设备又包括两种：直通设备和虚拟的外围设备。当一个直通设备被分配给虚拟机以后，该设备就会被映射到 IPA 地址空间，然后 VM 通过 IPA 直接访问物理设备。而当虚拟机需要使用虚拟的外围设备时，在地址转换的阶段-2，也就是从 IPA 转换到设备空间的时候会触发错误。当错误被 Hypervisor 捕获后由 Hypervisor 对设备进行模拟。

在实际的使用中，会给每个虚拟机都分配一个 ID 称为 VMID，用以标记特定 TLB 项和 VM 之间的对应关系。这样不同的 VM 就可以使用同一块 TLB 缓存。除此之外，TLB 也可以使用 ASID(Address Space Identification，地址空间标识符)来标记。每个应用分配一个 ASID，使得不同的应用之间也可以共享同一块 TLB 缓存。

VMSA 中支持三种类型的地址：VA、IPA 以及 PA。在 AArch64 执行模式下，内存访问的地址 VA 共 64 位，但查询页表时，仅使用其中的 48 位，故虚拟地址空间共 48 位。有了 Intel 架构中用户、内核页表隔离需要软件实现的前车之鉴[①]，VMSA 为 VA 提供两套页表，支持两个 VA 地址范围的地址翻译。由于查询页表仅使用 VA 中的前 48 位，剩余的 16 位可以用于标记属于两个 VA 地址范围中的哪一个。其中，在内核 VA 的范围内(Kernel Space)，所有 VA 的高 16 位均设为 1，故内核可以使用的范围是 0xffff000000000000～0xffffffffffffffff；在用户 VA 的范围内(User Space)，所有 VA 的高 16 位均设为 0，故用户空间可以使用的范围是 0x0000000000000000～0x0000ffffffffffff。

① 即 KPTI 补丁，由 Linux 内核实现，为了修复 Intel x86 处理器的 Meltdown 漏洞，性能开销较大。

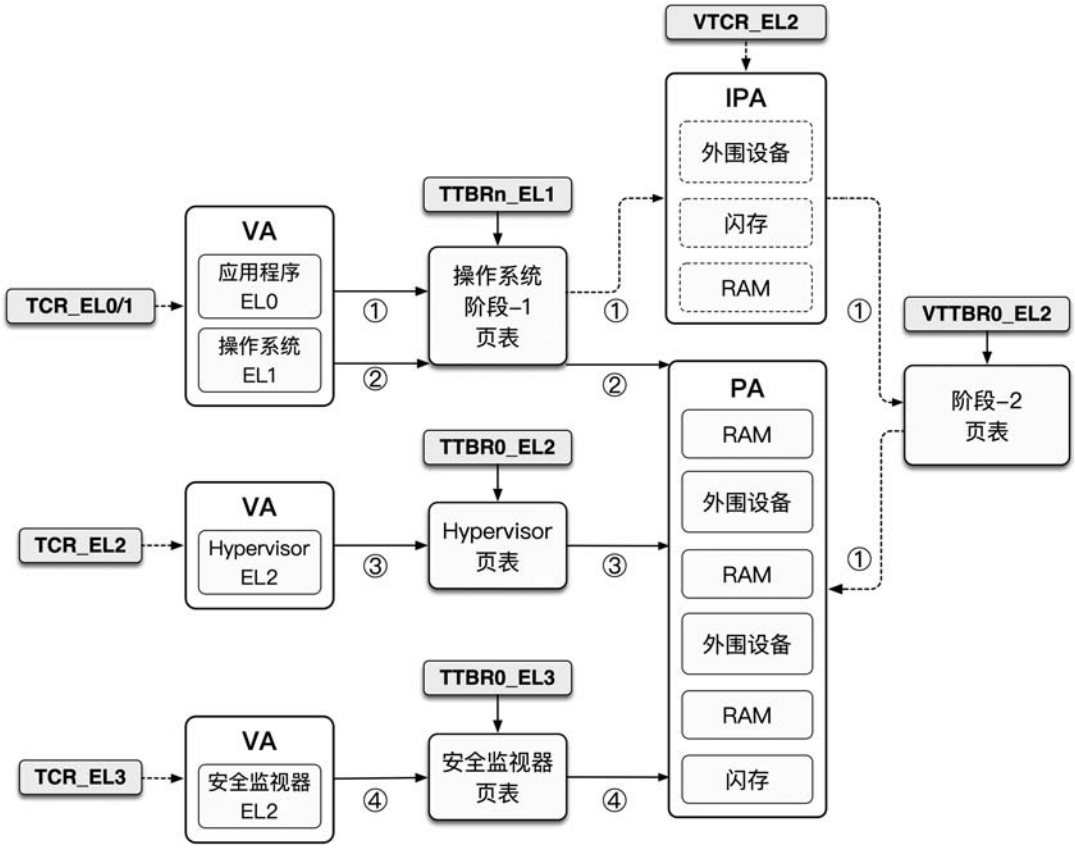
VMSA为内核VA空间和用户VA空间都提供了一个页表基地址寄存器，分别命名为TTBR1_ELx和TTBR0_ELx，即翻译表基地址寄存器(Translation Table Base Register)，后缀代表在ELx异常级下的PE有权限操作。这样，每当PE访问了一个VA，首先需要确定其高16位是否为0，MMU才能确定使用哪个寄存器作为页表基地址寄存器。然而只有EL0和EL1拥有两段VA空间，并具有TTBR{0,1}_EL{0,1}；对于EL2和EL3而言，只有较低段的VA空间，以及TTBR0_EL{2,3}，故EL2、EL3只能访问0x0000000000000000～0x0000ffffffffffff的VA。

如5.1.2节所述，由于ARM提供了EL0、EL1、EL2的异常级模式，操作系统运行在EL1，Hypervisor运行在EL2，这样软件上只能实现Type Ⅰ类型的Hypervisor。而依赖于Linux内核的KVM这类Type Ⅱ类型的Hypervisor将无法正确运行在EL2，于是只能实现为上层监控器结合底层监控器的模式，这样上下文切换会过于频繁，影响性能。为此，ARMv8.1提出了VHE，使得宿主机操作系统可以经过最少的修改量即可运行在EL2。在内存管理方面，VHE为EL2引入了用户态、内核态两套页表，以及TTBR{0，1}_EL2。对于操作系统原有的访问TTBR{0，1}_EL1的代码，VHE将该访问重定向到TTBR{0，1}_EL2。于是操作系统可以在EL2中透明地访问EL1寄存器，减少了操作系统的复杂度。

介绍完阶段-1中的两段VA机制，下面继续介绍阶段-2页表与相关寄存器。根据上文对于翻译流程的介绍，当运行在EL2的Hypervisor开启了阶段-2的地址翻译时，所有EL0、EL1中的内存翻译均需要两个阶段，其中阶段-1完成VA到IPA的翻译，使用的页表基地址寄存器为TTBR{0，1}_EL{0，1}，阶段-2完成IPA到PA的翻译，使用的寄存器为VTTBR0_EL2，保存着第二层页表的基地址。与Intel架构相同，两阶段地址翻译使用的两层页表在TLB不命中时也会产生大量的内存访问。

ARMv8中实现的翻译流程以及使用的页表基地址寄存器如图5-10所示。其中，有两个阶段的翻译流程使用了TTBR{0，1}_EL1以及VTTBR0_EL2，可以实现内存虚拟化；单阶段的翻译流程仅使用TTBR{0，1}_EL{0，1}以及TTBR0_EL{2，3}，可以实现虚拟内存以及用户、内核虚拟内存隔离。在EL2运行的Hypervisor将HCR_EL2的第0位设置为1时，则开启了第二阶段地址翻译，此后在EL0和EL1执行的访存指令中的VA都要经过两阶段的地址翻译，从而获得对应的IPA。在开启第二阶段地址翻译之前，Hypervisor需要将第二阶段页表的基地址写入VTTBR0_EL2中，从而正确地完成第二阶段地址翻译。

除了页表基地址寄存器外，ARMv8还提供了翻译控制寄存器用于控制MMU地址翻译的行为。阶段-1的地址翻译由TCR_EL{0，1，2，3}(Translation Control Register，翻译控制寄存器)控制，而阶段-2的地址翻译由VTCR_EL2(Virtual Translation Control Register，虚拟翻译控制寄存器)控制。这些地址翻译控制寄存器的作用包括指定当前的ASID/VMID、地址翻译的粒度大小以及各类地址的宽度，详细内容请查阅ARMv8架构说明手册。



注：①VA→IPA→PA；②③④VA→PA。

图 5-10　ARM KVM 架构图

5.4.3　内存属性、访问权限与缺页异常

对于所有的翻译流程，完成页表的查询后，可以得到三个结果：VA 对应的 PA、PA 对应的内存区域的**内存属性**以及**访问权限**，具体如下所述。

（1）访问权限大致包括可读、可写和可执行，这和 Intel x86 架构中页表项包含的访问权限类似。其区别是，每个访问权限是针对一个特定的异常级的，如标记为只能被运行在 EL2 的 PE 执行的内存区域，则不能被 EL0 和 EL1 异常级执行。

（2）内存属性包括对缓存行为的控制信息以及内存类型的信息，其中缓存行为包括：该段内存对应的缓存采用写穿（Write-Through）方式更新、或使用写回（Write-Back）方式更新、或该段内存不可缓存。

（3）内存类型包括普通内存，包括共享（Sharable，可在多个 PE 间共享）和不可共享（Non-sharable，只能被单一的 PE 访问）的普通内存，它支持预取、乱序访问、非对齐访问。例如，RAM、闪存等均属于普通内存。另一种类型是设备内存，即映射到物理地址空间内的

外围设备内存，或称为 MMIO 对应的物理内存区域。在图 5-10 中，外围设备代表了这类内存区域：它不支持预取、乱序访问、非对齐访问。回忆在第 3 章讲述 QEMU 内存虚拟化源代码时的 MemoryRegion，ARM 中实现的不同内存区域的内存类型和 MemoryRegion 的类型相似，即分为普通内存和设备内存。其他的内存属性和访存顺序相关，ARM 中实现了一种较弱的内存模型，需要控制 PE 的访存顺序，这部分内容较为复杂，本节不做讲述。

下面介绍地址翻译时可能出现的缺页异常。当 MMU 查询页表时，也会存在查询不成功的情况，此时会产生 MMU 异常，包括同步异常和异步异常。产生异常后，CPU 将跳转到相关的异常处理函数中，可以在处理函数中完成页表的填充。发生异常的原因将会记录在 ESR_EL{0，1，2}(Exception Syndrome Register，异常特征寄存器)中，它提供了异常的类型等信息，供软件使用。除 ESR 外，系统中还有 FAR_EL{0,1,2}(Fault Address Register，错误地址寄存器)，其作用和 Intel x86 架构下的 CR2 相同，它记录了导致缺页异常的 VA。

对于两阶段的翻译流程，MMU 异常可能发生在两个阶段中的任何一个阶段，而阶段-1 中引起的异常则会跳转到客户机异常处理函数，无须退出到 Hypervisor；阶段-2 中引起的异常则会退出到 Hypervisor，这和 Intel 架构下的扩展页表原理相类似。对于阶段-2 中的缺页异常，ARMv8 提供了 HPFAR_EL2(Hypervisor IPA Fault Address Register，Hypervisor IPA 错误地址寄存器)，它记录了引起阶段-2 缺页异常的 IPA，而 Intel 架构将引起 EPT 缺页异常的 GPA 记录在 VMCS 中，这有所不同。

第二阶段的缺页异常还被用来模拟 MMIO，详见 5.5.1 节的介绍。

5.4.4　MPAM

MPAM(Memory System Resource Partitioning and Monitoring，内存系统资源分割和监控)是一种通过确定性流控针对 CPU 访存系统资源隔离的技术手段，旨在解决大规模云部署时由于共享资源竞争带来的性能下降问题。下面将具体介绍 MPAM 在虚拟化中对于访存资源的隔离与应用。

MPAM 的系统架构可参考图 5-11，从图中可以看到，L3 缓存是被各个 CPU 所共享的。在 L3 高速缓存容量有限的情况下，如果某台虚拟机使用了过多的缓存，则会导致其他的虚拟机的缓存较少，使得其他虚拟机发送 TLB 命中失败的概率增大，从而影响其他虚拟机的访存性能。为了解决这个问题，在鲲鹏虚拟化中使用了 MPAM 来对访存资源进行隔离。MPAM 对访存的隔离有如图 5-12 所示的两种配置方式。

第一种(图 5-12(b))是通过优先级来配置，这种情况会给不同的虚拟机配置不同的优先级。优先级高的虚拟机可以优先取得对于共享缓存资源的使用权。这种配置方式可以确保运行重要任务的虚拟机能够优先使用共享缓存资源。

第二种(图 5-12(a))是以高速缓存的路(Cache Way)为粒度，使用位图来对资源进行分割，隔离不同虚拟机对于访存资源的使用。这也是目前鲲鹏 920 所使用的方案。

MPAM 在对不同业务流/虚拟机的访存控制上可以确保有一个明确的上下限。当虚拟机对访存资源的使用超过上限时则限制虚拟机的访存到上限以下；当虚拟机对于访存资

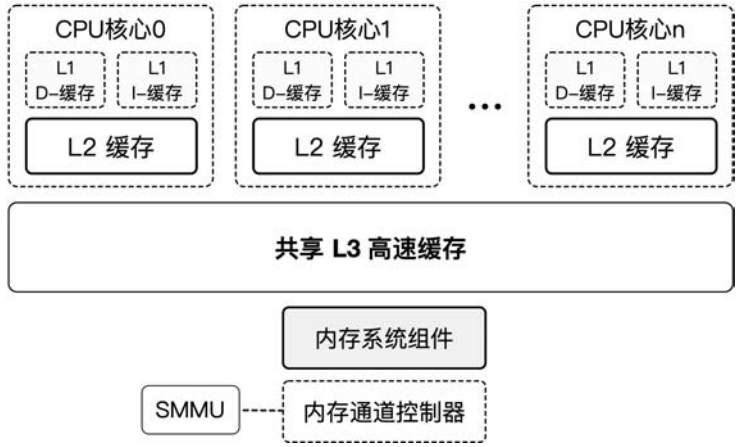


图 5-11　MPAM 系统框架

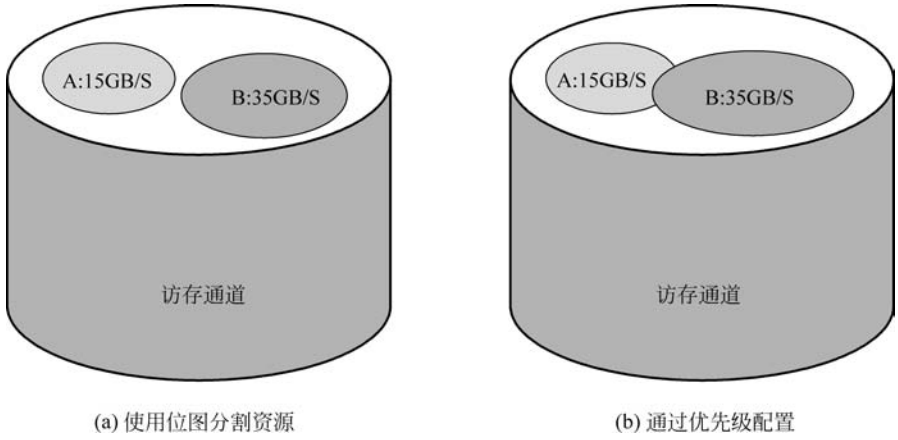


图 5-12　MPAM 配置方式

源的使用低于下限时则赋予其对于访存资源使用的优先权。在鲲鹏服务器的实际使用中，MPAM 能够有效降低在 CPU 访存过程中因虚拟机竞争带来的性能下降。

5.5　鲲鹏 I/O 虚拟化

5.5.1　MMIO 的模拟

与 x86 架构不同的是，ARM 架构的 CPU 对外设的访问只存在 MMIO 这一种方式。在 ARM－V8 架构下，当虚拟机向设备发起 MMIO 访问请求时，由于物理内存空间对虚拟机透明，虚拟机使用的是中间物理地址(IPA)，该地址并不能直接用于外设的访问，此时就

需要 Hypervisor 的介入。Hypervisor 会对 IPA 进行阶段-2 的地址转换，将 IPA 转换为能够用于 MMIO 访问的真实物理地址。

虚拟机通常可以拥有两种类型的外设，一种是直接分配给虚拟机的物理设备，一种是 Hypervisor 提供的虚拟设备，Hypervisor 会采用不同的方式来实现对这两种设备的 MMIO 访问模拟。对分配给虚拟机的物理设备的 MMIO 访问的处理过程比较简单，如图 5-13 所示，阶段-2 的页表项中会包含物理设备的物理空间地址与虚拟机 IPA 之间的映射，运行在虚拟机中的驱动程序可以通过 IPA 直接访问该物理设备。与访问直接分配的物理设备不同的是，每次虚拟机向虚拟设备发起 MMIO 访问时，都会触发阶段-2 缺页异常，之后 Hypervisor 会在异常处理程序中对该 MMIO 访问进行模拟。

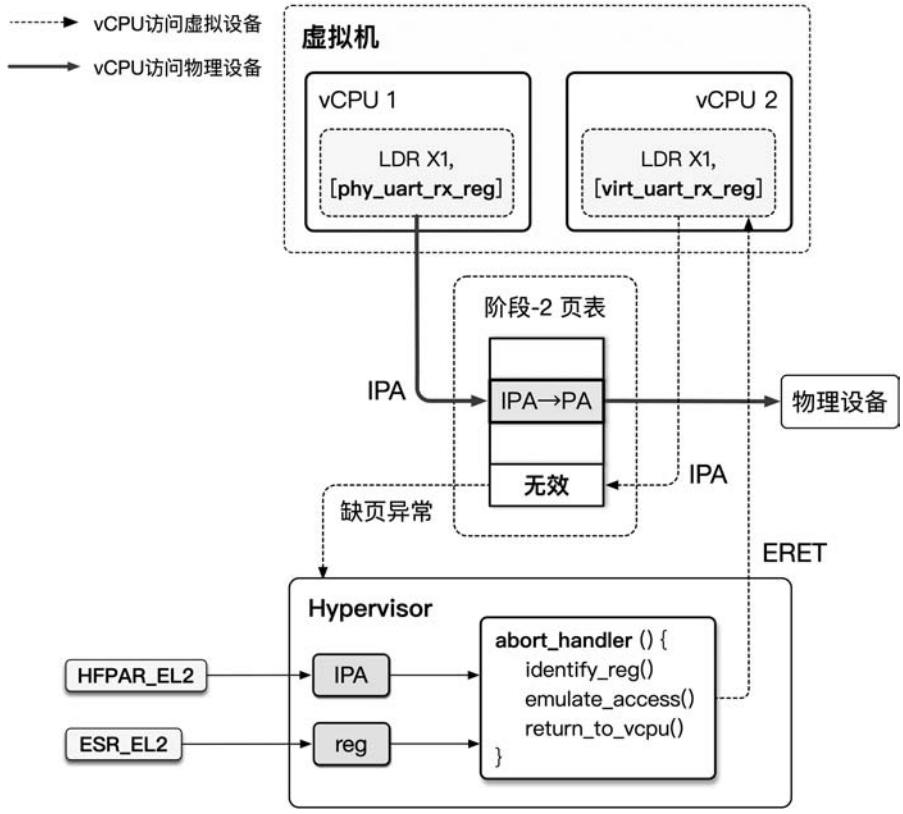


图 5-13　两种串口设备 MMIO 过程

在模拟 MMIO 访问之前，Hypervisor 需要知道虚拟机访问的具体虚拟设备，并定位该设备中被访问的寄存器，同时 Hypervisor 还需要知道与访问相关的信息，例如方式是读还是写、访问的大小以及用于传输数据的寄存器。图 5-13 展示了虚拟机访问虚拟串口设备的模拟过程，具体如下所述。

(1) vCPU 执行指令 LDR x0，[virt_uart_rx_reg]，向虚拟串口发起 MMIO 读访问。

(2) 该访问在阶段-2 翻译过程中会产生缺页异常，并触发中止(abort)异常。中止异常

会将 IPA 地址填充到 HPFAR_EL2 寄存器中，并将上文提到的访问相关信息[Read，4 bytes，x0]填充到 ESR_EL2 寄存器中。

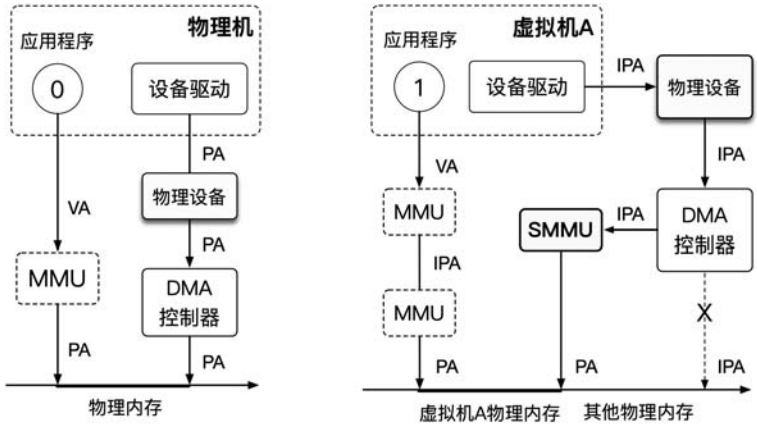
（3）由于 Hypervisor 负责分配和管理虚拟机的中间物理地址空间，Hypervisor 可以通过 HPFAR_EL2 中记录的 IPA 来确定 vCPU 访问的虚拟设备。Hypervisor 还可以通过特定函数获取 ESR_EL2 中记录的用于模拟 MMIO 访问的相关信息，如图 5-13 中的 identify_reg 函数会返回访问的虚拟设备寄存器。接下来，Hypervisor 会利用 IPA 信息和寄存器信息调用 emulate_access 函数，完成 MMIO 的模拟。之后，Hypervisor 通过 ERET 指令将控制流返回给 vCPU，vCPU 会继续执行下一条指令。

5.5.2 DMA 重映射——SMMUv3

常用的 I/O 虚拟化框架主要有两种：一种是半虚拟化场景下使用的 virtio 框架，该虚拟化框架是一种纯软件实现，与硬件无关，前面章节已经讨论过，这里就不再赘述；另一种就是设备直通的虚拟化方案。设备直通的一个关键点就是要解决 DMA 重映射问题，本节主要介绍 ARM 平台中的 SMMU(System Memory Management Unit，系统内存管理单元)。

在非虚拟化环境下，运行在内核态的设备驱动程序通过 DMA 相关的 API 接口完成 DMA 数据传输。在发起 DMA 请求时，驱动程序会在操作系统层面对访问的内存地址加以限制，确保应用程序内存访问的安全性。

然而在虚拟化环境下，虚拟机内的设备驱动可以与分配给该虚拟机的物理设备直接交互，但是驱动程序和物理设备却拥有不同的内存视图，驱动程序会将中间物理地址空间视为“真实”的物理地址空间，同时物理设备访问的则是实际主机物理地址空间，这就会给 DMA 传输带来问题。如图 5-14 所示，由于 DMA 控制器并不受内存阶段-2 翻译控制，物理设备会将驱动程序传入的 IPA 当作主机物理地址进行 DMA 传输，导致的后果是虚拟机可能会读写到属于其他虚拟机的物理内存地址甚至是 Hypervisor 所在的物理内存区域，破坏了系统的安全性和隔离性。



(a) 物理机的DMA　　(b) 虚拟机的DMA

图 5-14　非虚拟化与虚拟化环境下的地址翻译过程

为了解决 DMA 地址翻译问题，将 DMA 重映射到对应主机物理地址，各大厂商都推出了各自的硬件解决方案。4.2 节中介绍了 Intel 提出的 VT-d 技术，ARM 同样也提出了 SMMU。物理设备可以使用虚拟地址、IPA 或其他总线地址执行 DMA，SMMU 可以通过类似于 PE 中内存地址阶段-2 转换的方式将这些地址转换为 PA。

在 ARM 架构不断演进的过程中，ARM 推出了很多新的特性，为了支持这些特性，SMMU 也在不断演进。SMMUv1 主要支持 ARMv7 的页表格式，使用寄存器配置少量的设备流。SMMUv2 对 ARMv8.1-A 的页表格式提供了支持，并扩展了 SMMUv1，支持 64 位地址，同样使用寄存器配置少量的设备流。SMMUv1 和 SMMUv2 将传入的设备流映射到某个基于寄存器的上下文，该上下文会指向要使用的转换表和转换配置。该上下文还可以指示第二个上下文，用于阶段-2 的嵌套翻译。受寄存器数量的限制，使用基于寄存器的配置限制了上下文的可扩展性，并且不可能支持数千个并发的上下文。为了解决这一问题，SMMUv3 使用基于内存的配置结构。相较于使用寄存器，基于内存的配置结构可以支持大量的设备流，这也是 SMMUv3 相较于 SMMUv1 和 SMMUv2 最大的不同点。本节将对最新的 SMMUv3 进行介绍。

SMMU 的设计理念与 VT-d 有很多相似之处，在直通设备分配给虚拟机之前，Hypervisor 会为该设备建立阶段-2 地址翻译页表。阶段-2 地址翻译页表与 VT-d 中第二级(Second-level)地址翻译页表的作用类似，但 SMMU 与 VT-d 的不同之处是，SMMU 与 MMU 共用一套阶段-2 页表，而 VT-d 使用的是专用的 I/O 页表。Hypervisor 在阶段-2 地址翻译页表中会建立 IPA 与主机 PA 的映射关系，并限制设备所能访问的物理地址范围。图 5-15 展示了与 SMMU 相关的数据结构以及各配置结构之间的关系，下面对各配置结构分别进行介绍。

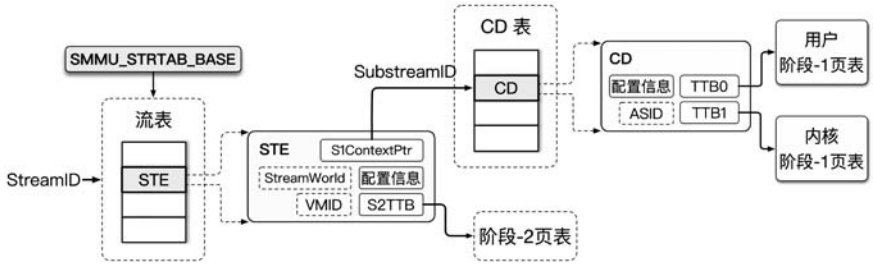


图 5-15　SMMU 使用的配置结构示例

SMMU 内有一个名为流表(Stream Table)的表，用于记录与每个设备阶段-1 和阶段-2 翻译页表基地址相关的信息。流表由 Hypervisor 维护，并且 Hypervisor 会将流表的基地址保存在 SMMU_STRTAB_BASE 寄存器中。流表由多个 STE(Stream Table Entry，流表项)构成，每个表项 STE 会记录一个发起 DMA 传输的设备的相关信息。值得注意的是，在 SMMU 中，阶段-1 转换和阶段-2 转换的使用是相互独立的，即 SMMU 可以只进行某一阶段翻译，也可以同时进行两个阶段翻译，这由 STE 中保存的配置信息决定。

STE 中有三个主要的成员：VMID、S2TTB(Stage-2 Translate Table Base，阶段-2 转换

表基)和 S1 上下文指针(S1ContextPtr)。VMID 属性代表设备所属的虚拟机。S2TTB 属性会指向此设备所属虚拟机的阶段-2 转换页表基地址。由于多个设备可能会同属于一个虚拟机,因此多个 STE 可以共享同一个阶段-2 转换表。此外,虚拟机中的多个进程可能同时使用虚拟机的某个设备,设备需要获得当前与自己交互的进程的相关信息,例如每个进程各自的阶段-1 页表。CD(Context Descriptor,上下文描述符)正是为描述进程信息而引入的数据结构。CD 中的 ASID 属性用于标识进程的地址空间,TTB0 和 TTB1 分别保存用户空间阶段-1 页表基地址和内核空间阶段-1 页表基地址(与 AArch64 中的寄存器 TTBR0 和 TTBR1 类似)。同一设备对应的全部 CD 构成了一个 CD 表,STE 中的 S1ContextPtr 成员会保存一个指向该 CD 表基地址的指针。

SMMU 使用流 ID(StreamID)来索引流表,并规定 StreamID 的大小为 0～32 位,但是 StreamID 使用的位数以及 StreamID 的构成都要依据具体实现决定。对于 PCI 设备来说,一般情况下 SMMU 会使用 PCI 设备标识符来填充 StreamID 的低 16 位,如果系统中存在多个根组件(Root Complex),则会使用高于 16 的位来扩展现有的 16 位 StreamID。在开启阶段-1 地址转换的情况下,SMMU 规定使用下一级流编号(SubstreamID)识别发起 DMA 请求的进程,并用于索引上下文描述表来选择使用的阶段-1 翻译页表。与 StreamID 类似,SubstreamID 被 SMMU 限定了大小范围为 0～20 位,SubstreamID 使用的位数以及 StreamID 的构成同样也要依据具体实现来决定。在 PCIe 系统中,SubstreamID 等价于 PASID,这与 VT-d 的可扩展模式中使用 PASID 获取第一级(First-level)翻译页表类似。

SMMU 中允许流表拥有如图 5-16 和图 5-17 所示的两种不同类型的组织形式。图 5-16 展示了所有 SMMU 都支持的流表的线性结构。线性流表是一个连续的 STE 数组,通过 StreamID 从 0 进行索引。线性流表包含 2n 个 STE,其中 n 最大可取到 SMMU 中支持的 StreamID 位数。图 5-17 展示了 StreamID 为 10 位这一条件下的两级流表结构。其中第一级流表包含多个描述符,使用 StreamID 的最高两位[9：8]检索第一级流表,每个描述符指向包含 STE 的第二级线性流表,StreamID 的低 8 位[7：0]用于索引第二级流表。每个第二级流表包含的 STE 数量可以根据需要灵活配置,以达到减少连续内存空间占用的目的。当线性流表中包含超过 64 个 STE 时,流表大小会超过 4KB,这意味着无法将流表存放在一个单独的内存页。所以 SMMU 中规定,当 StreamID 的位数大于 6 位时,必须使用两级流表结构。

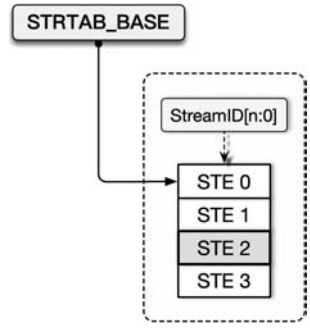


图 5-16 线性流表结构

与流表类似,SMMU 同样支持线性 CD 表和两级 CD 表结构。如图 5-18 所示,在使用两级 CD 表的情况下,STE 中的 S1ContextPtr 指向的是由多个 L1CD(Level 1 Context Descriptor,第一级上下文描述符)组成的第一级 CD 表。

SMMU 使用子流 ID(SubstreamID)的高位数据索引第一级 CD 表中的 L1CD。每个 L1CD 中的 L2Ptr 成员会指向某个第二级线性 CD 表的基地址。SMMU 使用 SubstreamID 的低位数据索引第二级线性 CD 表中的 CD,进而获取阶段-1 的地址转换页表。

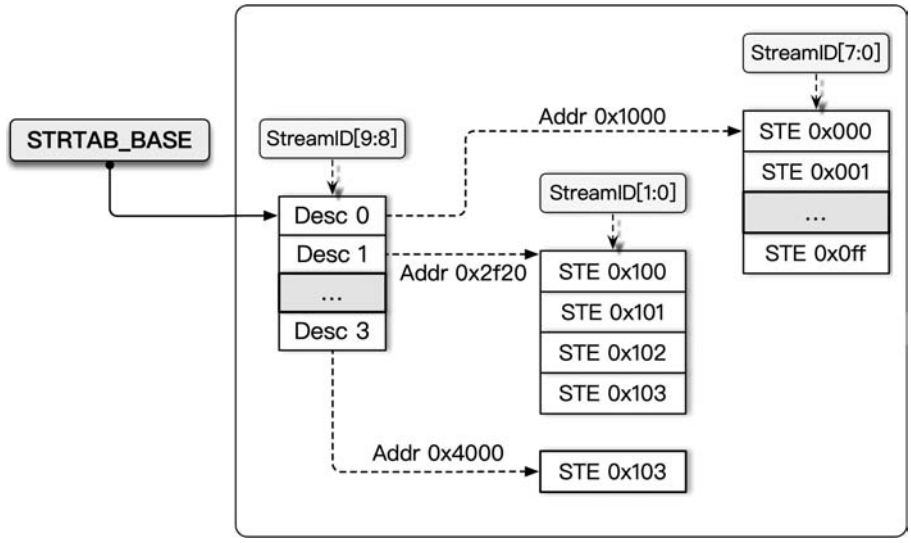


图 5-17　两级流表结构

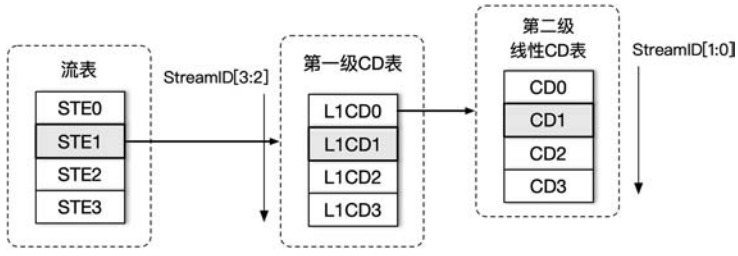


图 5-18　两级 CD 表

5.5.3　SMMUv3 中的缓存机制

4.2.4 节中介绍了 Intel VT-d 技术通过引入翻译缓存(Translation Cache)机制缓存与重映射地址转换相关的数据结构来加速地址转换过程,ARM 在 SMMU 中同样也引入了缓存机制。

SMMU 中 TLB 机制与前文介绍的 MMU 中的 TLB 机制类似,但不同之处在于 SMMU 查询 TLB 的过程除了 VMID、ASID、地址之外,还需要流世界(Stream World)参与。流世界主要用来描述设备数据流的安全状态和控制设备的进程所处的异常级。流世界的引入可以区分在不同异常级上运行的以及拥有不同安全状态的进程在 SMMU 中对应的 TLB 条目。例如,在输入地址都为 0x1000、ASID 都等于 3 的情况下,流世界可以判断发起本次访问的是运行在 EL1 中的安全进程还是非安全进程,进而查找到不同的 TLB 条目。关于流世界的更详细介绍,读者可以自行查阅 ARM 技术手册。

如图 5-19 展示了 SMMU 中在 TLB 参与下的地址翻译过程。该过程主要包含配置信息读取(步骤 1～3)和 DMA 地址翻译(步骤 4～5)两个阶段，流程如下。

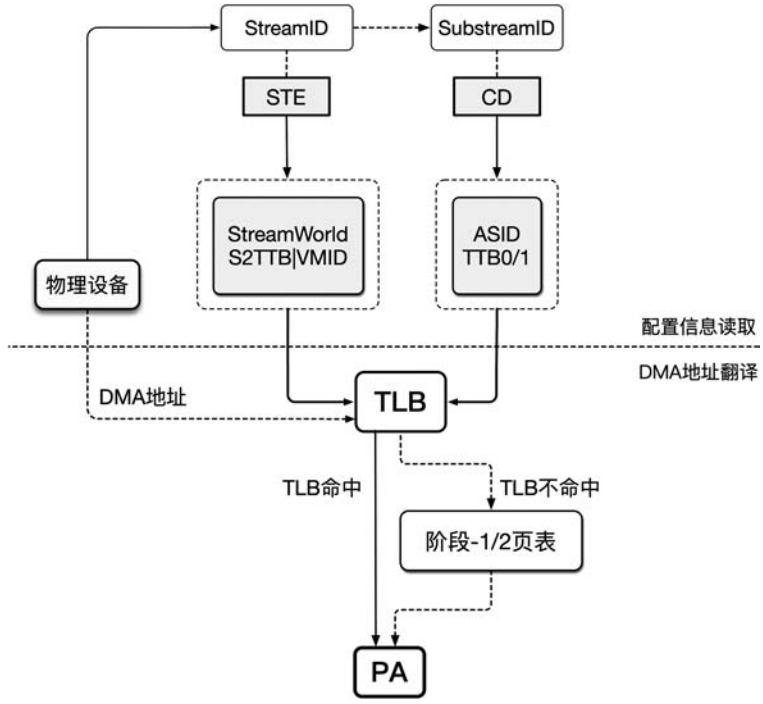


图 5-19　SMMU 地址翻译过程

(1) SMMU 从 I/O 事务中获取设备标识符，即 StreamID。

(2) SMMU 从 SMMU_STRTAB_BASE 寄存器中获取流表的基地址，并通过 StreamID 获取对应的 STE。

(3) 在开启阶段-1 转换的情况下，通过 SubstreamID 定位到对应的 CD，进而获取 ASID 和阶段-1 页表基地址。在开启阶段-2 转换的情况下，在 STE 中获取 VMID 和阶段-2 页表基地址以及流世界配置信息。

(4) SMMU 根据 DMA 地址、ASID、VMID 和流世界查询 TLB。如果 TLB 命中，可以直接获得目标物理地址以及访问权限信息。如果 TLB 未命中，则需要根据 DMA 地址通过相应地址翻译过程获得对应的目标物理地址，并将映射关系填充到 TLB 中。

(5) 设备根据目标物理地址进行数据传输。

SMMU 中缓存机制的引入使得在 TLB 命中的情况下，SMMU 可以直接从 TLB 中获取目标物理地址，同时并不需要经历“漫长”的阶段-1 和阶段-2 页表查询过程，从而提升 SMMU 的地址转换效率。

5.6　鲲鹏时钟虚拟化

在操作系统和应用程序中，对于时间的获取是非常必要且频繁的操作。如操作系统需要获取时间来提供日期显示、需要通过计时器来进行进程之间的调度、也需要通过周期性的时钟信号来实现看门狗功能等，所以时钟虚拟化尤为重要。下面将通过对比 x86 平台和鲲鹏的时钟虚拟化来理解时钟虚拟化。

与 x86 平台不同，ARM 平台对于时钟的虚拟化设计更为灵活。ARM 的时钟名称为 ARM 通用计时器(ARM Generic Timer)。该计时器直接加入了对时钟虚拟化的支持，由两部分组成：一个是由多个处理器共享的位于 SoC(System on Chip，片上系统)的系统计数器(System Counter)；另一个是每个处理器上的计时器。通用计时器由一系列的比较器组成，与公共的系统计数器进行比较。当比较器中的数据小于或者等于系统计数器的时候，比较器就会产生一个中断。图 5-20 展示了鲲鹏的通用计数器的组成部分和工作原理。

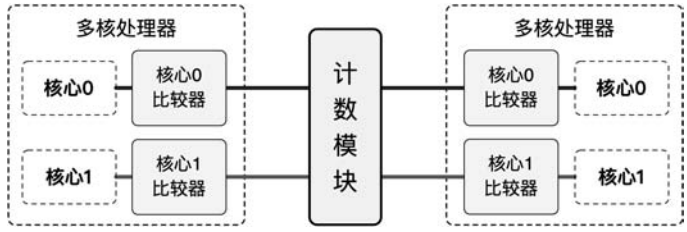


图 5-20　鲲鹏通用计数器组成

由于在实际使用中可能存在一个物理机上运行多个虚拟机的情况，当进行 vCPU 调度的时候，被调度的 vCPU 就会处于挂起状态，那么处于挂起状态的 vCPU 是如何计算并获取实时时钟的呢？

假设 Hypervisor 调度 vCPU 的时间可以忽略不计。如图 5-21 所示，在 4ms 的时间里，vCPU0 和 vCPU1 各自运行了 2ms。但是如果 vCPU0 在 T＝0 时刻设置了一个 3ms 的计时器，而此时 Hypervisor 已经通过调度将运行环境和运行权限交给了 vCPU1，那么此时的中断将会如何触发？除此之外还有一个问题：由于 Hypervisor 的调度，vCPU0 和 vCPU1 各自只有一半的时间在使用 CPU 资源，那么设置的 3ms 计时器是 vCPU0 里的虚拟 3ms 还

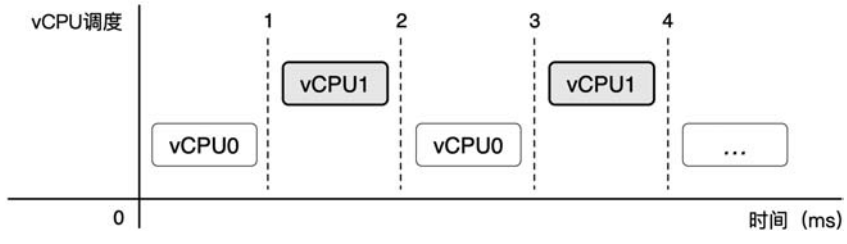


图 5-21　vCPU 调度

是实际中的 3ms(也就是 wall-clock 时间里面的 3ms)?

为了支持虚拟化,ARM 又加入了两个新的概念：虚拟计数器(Virtual Counter)和虚拟计时器(Virtual Timer)。也就是说,在鲲鹏中,既有物理计数器(Physical Counter)又有虚拟计数器。值得注意的是,虽然名为虚拟计数器,但是该计数器确是实际存在的计数器而不是软件模拟。由于是两种不同的计数器,所以在访问计数器的数据的时候所读取的寄存器也不同。对于物理计数器,读取数据时要访问的寄存器为 CNTPCT,虚拟计数器对应的寄存器则为 CNTVCT。vCPU 中的程序可以访问两个计数器：EL1 物理计数器和 EL2 虚拟计数器。其中 EL1 的物理计数器与系统计数器产生的计数做比较,虚拟计数器则是物理计数器减去一个偏移量。由于这个偏移量保存在寄存器中,Hypervisor 通过寄存器来读取偏移量的大小,这样就可以在 vCPU 被调度器挂起的时候将时间透传给 vCPU。虚拟计数器和物理的计数器关系如图 5-22 所示

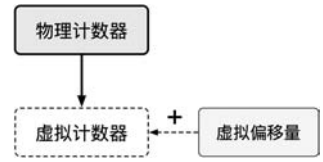


图 5-22 虚拟计数器与物理计数器关系

那么这样的设计在实际调度中是如何工作的呢？可以用图 5-23 来表示。

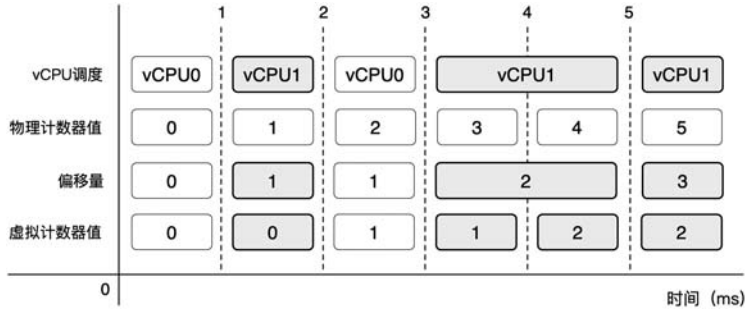


图 5-23 时钟虚拟化在 vCPU 调度时的工作原理

这幅图清晰地展示了鲲鹏使用虚拟计数器来保证多个虚拟机计时器时间同步的工作原理。如图 5-23 所示,第一行展示了两个虚拟 CPU：vCPU0 和 vCPU1 的实际调度时间；第二行展示了物理计时器以 1ms 的周期记录时间的流逝；第三行则是偏移量的值,由于是两个虚拟的 CPU,所以需要两个不同的偏移量：vCPU0 的 offset0(图 5-23 中偏移量中浅色的值)和 vCPU1 的 offset1(图 5-23 中偏移量中深色的值)。在 T=0 时刻,由于虚拟机刚刚启动,所以 offset0 和 offset1 均为 0。T=1 时切换到 vCPU1 运行,由于第 1ms 都是 vCPU0 在占用,所以 offset1 的值为 1。T=2 时由于 vCPU1 也只运行了 1ms ,所以 offset0 的值为 1,后面的 offset 同理。

有了偏移量的存在,vCPU 在切换的时候也可以获取到实时的时间,保证了多个 vCPU 运行场景中对于时间的正确获取。

这种使用虚拟计数器的设计给鲲鹏的时钟虚拟化带来了较强的竞争力。在实际使用中,Hypervisor 会被配置去使用物理计时器,而虚拟机则被设置去使用虚拟计时器。这样就可以使 Hypervisor 和虚拟机使用不同的计时器,在使用中就不存在冲突了。在虚拟机使

用计时器的时候，不用通过陷出到 EL2 异常级访问计时器的数值，减少了异常级切换带来的损耗，提高了时钟虚拟化的效率。但是值得注意的是，如果在一个平台上运行了多台虚拟机，在虚拟机切换的时候还是要将虚拟计时器的数据保存起来。

下面结合代码深入分析鲲鹏虚拟化的实现。

首先是虚拟时钟的初始化，初始化动作在内核启动时进行。初始化时会创建一个 arch_timer_kvm_info 结构体来记录中断和计时器，代码如下[①]。

linux - 5.10/include/clocksource/arm_arch_timer.h

```
struct arch_timer_kvm_info {
    struct timecounter timecounter;
    int virtual_irq;
    int physical_irq;
};
```

然后在 KVM 启动时，调用 kvm_timer_hyp_init 函数初始化中断号和读取虚拟计数器的函数。之后就是在 QEMU 创建虚拟机时调用 kvm_timer_init 函数来读取 CPU 的虚拟计数器，重点是初始化 vCPU。上面提到过，在多个虚拟机的场景下，如果需要切换虚拟机，就要对时钟的上下文环境进行保存。所以在初始化 vCPU 时，每个 vCPU 都会为自己维护一个和时钟虚拟化相关的结构体，用来保存时钟虚拟化的上下文环境，代码如下。

linux - 5.10/include/kvm/arm_arch_timer.h

```
struct arch_timer_cpu {
    struct arch_timer_context timers[NR_KVM_TIMERS];
    /* Background timer used when the guest is not running */
    struct hrtimer          bg_timer;
    /* Is the timer enabled */
    bool            enabled;
};
```

然后在第一次运行 vCPU 的时候通过调用 vgic 函数创建中断映射，此时时钟虚拟化的初始化动作完成。完成初始化之后虚拟时钟就可以正常工作了。

本章小结

本章主要围绕鲲鹏平台介绍了 ARM 虚拟化的一些相关原理，包括 CPU、内存、I/O、中断以及时钟虚拟化。前面的第 2、3、4 章分别介绍了 CPU、内存和 I/O 在 x86 下的虚拟化实现以及 x86 和 ARM 下的通用实现方式，本章则着重介绍鲲鹏平台中 ARM 虚拟化特有的虚拟化技术，如 CPU 部分的 GIC 中断、内存的阶段-2 转换、I/O 部分的 SMMU 以及鲲鹏的时钟虚拟化。

① Linux kernel 5.10 源码下载地址：https://github.com/torvalds/linux，下载时须选用 v5.10 标签。