CHAPTER 3

第 3 章 虚拟化概述

通过前面章节的介绍，了解到虚拟化技术的历史与背景知识，从这章开始，将进一步揭开 VMM 神秘的面纱，对其内部实现的基本原理作一番全面扫描。

传统的虚拟化技术一般是通过陷入再模拟的方式实现的，而这种方式依赖于处理器的支持。也就是说，处理器本身是否是一个可虚拟化的体系结构。所以本章首先从可虚拟化结构的定义入手，介绍 VMM 实现中的一些基本概念。显然，某些处理器在设计之初并没有充分考虑虚拟化的需求，而不具备一个完备的可虚拟化结构。如何填补这些结构上的缺陷，直接促使了本书提到的三种主要虚拟化方式的产生。

不论采取何种虚拟化方式，VMM 对物理资源的虚拟可以归结为三个主要任务：处理器虚拟化、内存虚拟化和 I/O 虚拟化。本章前面部分就围绕这三个部分展开介绍虚拟化的基本原理，对于不同虚拟化方式的实现细节，本书后续章节会有详细的描述。本章后面部分着重介绍 VMM 的功能、组成和分类，并且对目前市场上流行的虚拟化产品及其特点做一些简单的介绍，使读者对现阶段典型的虚拟化产品有一些了解。

3.1 可虚拟化架构与不可虚拟化架构

一般来说，虚拟环境由三个部分组成：硬件、VMM 和虚拟机，如图 3-1 所示。在没有虚拟化的情况下，操作系统直接运行在硬件之上，管理着底层物理硬件，这就构成了一个完整的计算机系统，也就是下文所谓的“物理机”。在虚拟环境里，虚拟机监控器 VMM 抢占了操作系统的位置，变成了真实物理硬件的管理者，同时向上层的软件呈现出虚拟的硬件平台，“欺骗”着上层的操作系统。而此时操作系统运行在虚拟平台之上，仍然管理着它认为是“物理硬件”的虚拟硬件，俨然不知道下面发生了什么，这就是图 3-1 中的“虚拟机”。

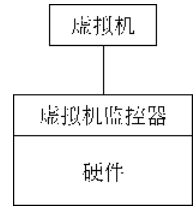

图 3-1 虚拟环境的组成

虚拟机可以看作是物理机的一种高效隔离的复制，上面的定义里蕴涵了三层含义（同质、高效和资源受控），这也是一个虚拟机所具有的三

个典型特征。

关于这三个特点的含义，在第 1 章已经阐述。

或许硬件体系结构会有所限制，或许 VMM 的实现方式会有所不同，但如果符合不了上面的三个特点，那么可以说，这个虚拟机是失败的，这个 VMM 的骗术是不高明的。

上面提到的虚拟机必须具备的三个特点也就决定了不是在任何体系结构下都可以虚拟化的。给定一个系统，其对应的体系结构是否可虚拟化，就要看能否在该系统上虚拟化出具备上面三种特征的虚拟机。

为了进一步研究可虚拟化的条件，先从指令开始着手介绍。

大多数的现代计算机体系结构都有两个或两个以上的特权级，用来分隔系统软件和应用软件。系统中有一些操作和管理关键系统资源的指令会被定为特权指令，这些指令只有在最高特权级上能够正确执行。如果在非最高特权级上运行，特权指令会引发一个异常，处理器会陷入到最高特权级，交由系统软件来处理。在不同的运行级上，不仅指令的执行效果是不同的，而且也并不是每个特权指令都会引发异常。假如一个 x86 平台的用户违反了规范，在用户态修改 EFLAGS 寄存器的中断开关位，这一修改将不会产生任何效果，也不会引起异常陷入，而是会被硬件直接忽略掉。

在虚拟化世界里，还有另一类指令被称为敏感指令，简言之就是操作特权资源的指令，包括修改虚拟机的运行模式或者下面物理机的状态；读写敏感的寄存器或是内存，例如时钟或者中断寄存器；访问存储保护系统、内存系统或是地址重定位系统以及所有的 I/O 指令。

显而易见，所有的特权指令都是敏感指令，然而并不是所有的敏感指令都是特权指令。

为了 VMM 可以完全控制系统资源，它不允许直接执行虚拟机上操作系统（即客户机操作系统）的敏感指令。也就是说，敏感指令必须在 VMM 的监控审查下进行，或者经由 VMM 来完成。如果一个系统上所有敏感指令都是特权指令，则能够用一个很简单的方法来实现一个虚拟环境：将 VMM 运行在系统的最高特权级上，而将客户机操作系统运行在非最高特权级上，当客户机操作系统因执行敏感指令（此时，也就是特权指令）而陷入到 VMM 时，VMM 模拟执行引起异常的敏感指令，这种方法被称为“陷入再模拟”。

总而言之，判断一个结构是否可虚拟化，其核心就在于该结构对敏感指令的支持上。如果在某些结构上所有敏感指令都是特权指令，则它是可虚拟化的结构；否则，如果它无法支持在所有的敏感指令上触发异常，则不是一个可虚拟化的结构，我们称其存在“虚拟化漏洞”。

我们已经知道，通过陷入再模拟敏感指令的执行来实现虚拟机的方法是有前提条件的：所有的敏感指令必须都是特权指令。否则，要么系统的控制信息会被虚拟机修改或访问，要么 VMM 会遗漏需要模拟的操作，影响虚拟化的正确性。如果一个体系结构上存在敏感指令不属于特权指令，那么其就存在虚拟化漏洞。有些计算机体系结构是存在虚拟化漏洞的，就是说它们不能很高效地支持系统虚拟化。

虽然虚拟化漏洞有可能存在，但是可以采用一些办法来填补或避免这些漏洞。最简单最直接的方法是，如果所有虚拟化都采用模拟来实现，例如解释执行，就是取一条指令，模拟出这条指令执行的效果，再继续取下一条指令，那么就不存在所谓陷入不陷入的问题，从而避免了虚拟化漏洞。这种方法不但能够适用于模拟与物理机相同体系结构的虚拟机，而且也能模拟不同体系结构的虚拟机。虽然这种方法保证了所有指令（包括敏感指令）执行受到VMM的监督审查，但是它对每条指令不区别对待，其最大的缺点很明显就是性能太差，是不符合虚拟机“高效”特点的，导致其性能下降为原来的十分之一甚至几十分之一。

既要填补虚拟化漏洞，又要保证虚拟化的性能，只能采取一些辅助的手段。或者直接在硬件层面填补虚拟化漏洞，或者通过软件的办法避免虚拟机中使用到无法陷入的敏感指令。这些方法都不仅保证了敏感指令的执行受到VMM的监督审查，而且保证了非敏感指令可以不经过VMM而直接执行，从而相比完全解释执行来说，性能得到了极大的提高。

3.2 处理器虚拟化

处理器虚拟化是VMM中最核心的部分，因为访问内存或者I/O的指令本身就是敏感指令，所以内存虚拟化与I/O虚拟化都依赖于处理器虚拟化的正确实现。

3.2.1 指令的模拟

VMM运行在最高特权级，可以控制物理处理器上的所有关键资源；而客户机操作系统运行在非最高特权级，所以其敏感指令会陷入到VMM中通过软件的方式进行模拟。从客户机操作系统的角度而言，无论一条指令是直接执行在物理处理器上，还是被VMM软件模拟，其期望的执行效果必须正确。所以，处理器虚拟化的关键在于正确模拟指令的行为。

在介绍指令模拟之前，我们理解三个概念：虚拟寄存器、上下文和虚拟处理器。它们有助于理解虚拟处理器模拟指令。

从某种程度上来说，物理处理器无非包括了一些存放数据的物理寄存器，并且规定了使用这些寄存器的指令集，然后按照一段预先写好的指令流，在给定时间点使用给定的部分寄存器来完成某种目的。

在没有虚拟化的环境里，操作系统直接访问物理处理器，处在最高的特权级别，可以控制系统中的所有关键资源，包括寄存器、内存和I/O外设等。但是，当VMM接管物理处理器后，昔日的操作系统成为了客户机操作系统而降级到非最高特权级别上，这时，其试图访问关键资源的指令就成为了敏感指令。VMM会通过各种手段，保证这些敏感指令的执行能够触发异常，从而陷入到VMM进行模拟，以防止对VMM自身的运行造成破坏。

所以，当客户机操作系统试图访问关键资源的时候，该请求并不会真正发生在物理寄存器上。相反，VMM会通过准确模拟物理处理器的行为，而将其访问定位到VMM为其设计

与物理寄存器对应的“虚拟”的寄存器上。当然，从 VMM 实现来说，这样的虚拟寄存器往往是在内存中。

图 3-2 是一个具体的访问控制寄存器 CR0 的例子。当处理器取下一条指令 MOV CR0,EAX 后，发现特权级别不符合，则抛出异常，VMM 截获这个异常之后模拟处理器的行为，读取 EAX 的内容并存放到虚拟的 CR0 中。由于虚拟的 CR0 存放在 VMM 为该虚拟机设计的内存区域里，因此该指令执行的结果并不会让物理的 CR0 的内容发生改变。等到下一次，当虚拟机试图读 CR0 时，处理器也会抛出异常，然后由 VMM 从虚拟的 CR0 而不是物理的 CR0 中返回内容给虚拟机。

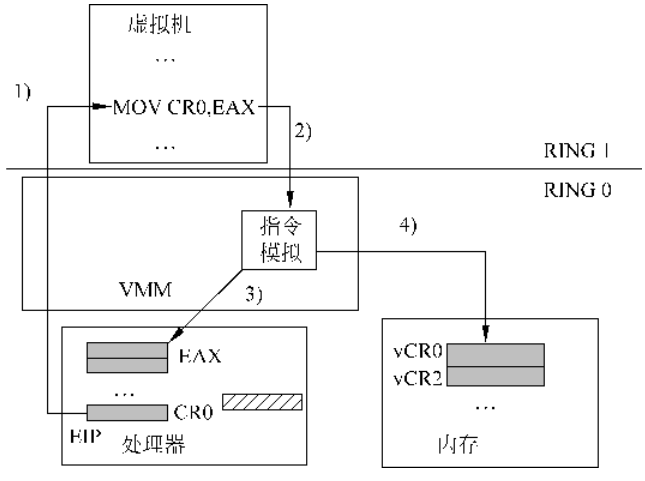

图 3-2 访问虚拟寄存器

在没有虚拟化的环境里，操作系统直接负责物理处理器管理，负责进程间调度和切换。但是，VMM 接管物理处理器后，客户机操作系统没有管理物理处理器的权利，可以说此时它已经运行在 VMM 为之设计的虚拟处理器之上，管理虚拟处理器，并在虚拟处理器上负责该虚拟机内进程间调度和切换。而 VMM 管理物理处理器，负责虚拟处理器的调度和切换，以保证在给定时间内，每个虚拟处理器上的当前进程可以在物理处理器上运行一段时间。但是，不管是何种调度切换，必然要涉及到保留现场，这个现场就是上下文状态，只不过前一种情况是进程上下文，后一种情况是虚拟处理器上下文。为了让读者更好地理解，我们从进程上下文开始类比地介绍。

通过第 2 章我们知道，在某个时刻，物理处理器中的寄存器状态构成了当前进程上下文状态。

进程上下文主要是与运算相关的寄存器状态，例如 EIP 寄存器指向进程当前执行的指令，ESP 存放着当前进程的堆栈指针等。当操作系统进行调度时，当前进程的上下文，即上述寄存器状态被保存在进程特定的内存区域中，而下一个进程的上下文被恢复到相应的寄

存器中，从进程角度看，就好像从未被中断一样。

虚拟处理器上下文比进程上下文更为复杂，因为客户机操作系统本身包含许多敏感指令，会试图访问和修改物理处理器上定义的所有寄存器，而这种访问和修改会被 VMM 重定位到虚拟处理器上。所以，对于虚拟处理器，其上下文包括了更多的系统寄存器，例如 CR0、CR3、CR4 和各种 MSR 等。当 VMM 在决定切换虚拟处理器的时候，为了让虚拟机看来好像也从未被中断过一样，VMM 需要考虑保存与恢复的上下文也更为复杂。

我们没有介绍虚拟处理器，但是上面已经谈到虚拟处理器，那什么是虚拟处理器呢？虚拟处理器其实也是一个虚拟的概念，一个逻辑上而非物理上的概念，可以从两个角度来理解。

首先，从客户机操作系统来说，其在运行的处理器需要具备与其“期望”的物理处理器一致的功能和行为，这种“期望”的前提条件甚至可以允许客户机操作系统的修改，例如 VMM 可以通过修改客户机操作系统的源代码，使客户机操作系统所“期望”的与 VMM 所呈现的功能集合一致。典型的“期望”包括：

（1）指令集合与执行效果。

（2）可用寄存器集合，包括通用寄存器以及各种系统寄存器。

（3）运行模式，例如实模式、保护模式和 64 位长模式等。处理器的运行模式决定了指令执行的效果、寻址宽度与限制以及保护粒度等。VMM 必须正确模拟虚拟机期望的运行模式，否则会对虚拟机甚至是 VMM 自身的运行产生严重影响。

（4）地址翻译系统，例如页表级数。

（5）保护机制，例如分段和分页等。

（6）中断/异常机制，例如虚拟处理器必须能够正确模拟真实处理器的行为，在错误的执行条件下，为虚拟机注入一个虚拟的异常。

其次，从 VMM 的角度来说，虚拟处理器是其需要模拟完成的一组功能集合。虚拟处理器的功能可以由物理处理器和 VMM 共同完成。对于非敏感指令，物理处理器直接解码处理其请求，并将相关的效果直接反映到物理寄存器上；而对于敏感指令，VMM 负责陷入再模拟，从程序的角度也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物理处理器的行为将效果反映到虚拟寄存器上。

值得一提的是，VMM 已经可以为虚拟机呈现出与实际物理机不一致的功能和行为。例如，虚拟处理器的个数，可以与物理处理器的个数不一致。在有多个物理处理器的平台上，VMM 可以让虚拟机看到该平台好像只有一个物理处理器（即一个虚拟处理器），而在只有一个物理处理器的平台上，VMM 可以让虚拟机看到该平台好像有多个物理处理器（即多个虚拟处理器），这种效果完全取决于用户对虚拟环境的配置，以及 VMM 自身的策略。

在介绍完以上三个概念之后，基本上也就了解了在处理器虚拟化中，不论是定义虚拟寄存器和虚拟处理器，还是利用上下文进行虚拟处理器调度切换，其宗旨都是让虚拟机里执行的敏感指令陷入下来之后，能被 VMM 模拟，而不要直接作用于真实硬件上。

当然，模拟的前提是能够陷入。接下来，理解一下客户机操作系统执行时，是如何通知VMM的，也就是VMM的陷入方式。概括地说，VMM陷入是利用了处理器的保护机制，利用中断和异常来完成的，它有以下几种方式。

(1) 基于处理器保护机制触发的异常，例如前面提到的敏感指令的执行。处理器会在执行敏感指令之前，检查其执行条件是否满足，例如当前特权级别、运行模式以及内存映射关系等。一旦任一条件不满足，VMM得到陷入然后进行处理。

(2) 虚拟机主动触发异常，也就是通常所说的陷阱。当条件满足时，处理器会在触发陷阱的指令执行完毕后，再抛出一个异常。虚拟机可以通过陷阱指令来主动请求陷入到VMM中去。例如，在后面介绍的类虚拟化技术就是通过这种方式实现Hypercall的。

(3) 异步中断，包括处理器内部的中断源和外部的设备中断源。这些中断源可以是周期性产生中断的时间源，也可以是根据设备状态产生中断的大多数外设。一旦中断信号到达处理器，处理器会强行中断当前指令，然后跳到VMM注册的中断服务程序，所以这也为VMM的陷入提供了一种途径。例如，VMM可以通过调度算法指定当前虚拟机运行的时间片长度，然后编程外部时钟源，确保时间片用完时触发中断，从而允许VMM进行下一次调度。

3.2.2 中断和异常的模拟及注入

中断和异常机制是处理器提供给系统程序的重要功能，异常保证了系统程序对处理器关键资源的绝对控制，而中断提供了与外设之间更有效的一种交互方式。所以，VMM在实现处理器虚拟化时，必须正确模拟中断与异常的行为。

VMM对于异常的虚拟化需要完全遵照物理处理器对于各种异常条件的定义，再根据虚拟处理器当时的内容，来判断是否需要模拟出一个虚拟的异常，并注入到虚拟环境中。

VMM通常会在硬件异常处理程序和指令模拟代码中进行异常虚拟化的检查。无论是哪一条路径，VMM需要区分两种原因：一是虚拟机自身对运行环境和上下文的设置违背了指令正确执行的条件；二是虚拟机运行在非最高特权级别，由于虚拟化的原因触发的异常。第二种情况是由于陷入再模拟的虚拟化方式所造成的，并不是虚拟机本身的行为。而对于第一种情况的检查，VMM实际是在虚拟处理器的内容上进行的，因为它反映了虚拟机所期望的运行环境。错误的异常注入会让客户机操作系统做出错误的反应，后果无法预知。

物理中断的触发来自于特定的物理中断源，同样，虚拟中断的触发来自于虚拟设备的模拟程序。当设备模拟器(后面会讲到)发现虚拟设备状态满足中断产生的条件时，会将这个虚拟中断通知给中断控制器的模拟程序，例如模拟LAPIC。最后，VMM会在特定的时候检测虚拟中断控制器的状态，来决定是否模拟一个中断的注入。而这里的虚拟中断源包括：处理器内部中断源的模拟，例如LAPIC时钟、处理器间中断等；外部虚拟设备的模拟，例如8254、RTC、IDE、网卡和电源管理模块等；直接分配给虚拟机使用的真实设备的中断，通常

来自于 VMM 的中断服务程序；自定义的中断类型。

不管怎样，当 VMM 决定向虚拟机注入一个中断或是异常时，它需要严格模拟物理处理器的行为来改变客户指令流的路径，而且还要包括一些必需的上下文保护与恢复。VMM 需要首先判断当前虚拟机的执行环境是否允许接受中断或是异常的注入，假如客户机操作系统正好通过 RFLAGS.IF 位禁止了中断的发生，这时 VMM 就只能把中断事件暂时缓存起来，直到某时刻客户机操作系统重新允许了中断的发生，VMM 才立即切入来模拟一个中断的注入。而当中断事件不能被及时注入时，VMM 还要进一步考虑如下因素。

(1) 该中断类型是否允许丢失中断，如果允许，VMM 则可以将其后到达的多个同类型中断合为一个事件；否则，VMM 必须要跟踪所有后续到达的中断实例，在客户指令流重新允许中断时，将每一个缓存的中断一一注入。

(2) 该中断在阻塞期间是否被中断源取消，这决定了 VMM 是否会额外地注入一个已经被取消的假中断。

(3) 当一次阻塞的中断实例比较多时，VMM 可能还需要考虑客户机操作系统能否处理短时间内大量同类型的中断注入，因为这在真实系统中可能并不出现。

当然，上面只是简单地列举了一些中断注入时需要考虑的因素，在实际的实现中基于正确性与效率方面还有更多需要考虑的因素，详细内容请参考后面的章节。

在模拟中断或异常的注入时，VMM 需要首先判断是否涉及到运行模式的切换。假如虚拟机可能运行在一个 64 位兼容模式，而其中断/异常处理函数运行在 64 位长模式，这时 VMM 就需要按照处理器的规定，将虚拟机的运行模式进行软件切换，对保存的客户上下文进行相应的修改。在可能的模式切换完成后，VMM 还需要根据真实处理器在该模式下的中断注入过程，完整地进行软件模拟。例如，将必需的处理器状态(指令地址、段选择子等)复制压入当前模式下对应中断/异常服务程序的堆栈；到中断模拟逻辑去查找发生中断的向量号；根据该向量号来查找相关的中断/异常服务程序的入口地址；最后修改虚拟机的指令地址为上述入口地址，然后返回到虚拟机去执行等。

总的来说，中断/异常的虚拟化由中断/异常源的定义、中断/异常源与 VMM 处理器虚拟化模块间的交互机制以及最终模拟注入的过程所组成。不同的 VMM 在这个方面的设计和实现都不尽相同，根据不同的客户操作系统的类型也会造成差异。同样，请详细阅读后面的章节来了解针对不同 VMM 模式的具体实现。

3.2.3 对称多处理器技术的模拟

在没有虚拟化的环境里，对称多处理器技术可以让操作系统拥有并控制多个物理处理器，它通过提供并发的计算资源和运算逻辑，允许上层操作系统同时调度多条基于不同计算目的的进程并发执行，从而有效地提高系统的吞吐率与性能。

同样道理，当物理计算资源足够多时，VMM 也可以考虑为虚拟机呈现出多个虚拟处理

器，也就是客户对称多处理器虚拟化技术，也称客户 SMP 技术。这样，当这些虚拟处理器同时被调度在多个物理处理器上执行时，也可以有效地提高给定虚拟机的性能。虽然从 VMM 调度的角度来说，因为 VMM 仍然在物理处理器上基于一定的策略来管理多个虚拟处理器，客户 SMP 功能的加入并没有带来整体性能改观。但是，对某个虚拟机来说，以前每个虚拟处理器属于各自不同的虚拟机，而现在客户 SMP 功能引入后，某些具有相同属性的虚拟处理器可以隶属于同一个虚拟机，从而使该虚拟机因为拥有多个计算资源而可以让其虚拟机性能较其他虚拟机得到提高，如图 3-3 所示。

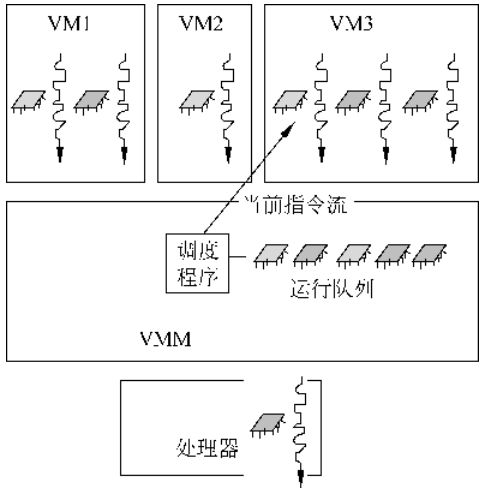

图 3-3 对称多处理器的模拟

因此可以说，客户 SMP 中虚拟处理器的数目与实际物理处理器数目之间没有必然联系，也就是说，客户 SMP 中虚拟处理器的个数可以小于、等于或是大于实际物理处理器个数。当然，客户 SMP 功能引入后，VMM 在虚拟环境的管理和责任上发生了一些变化。

首先，VMM 必须按照客户机操作系统期望的那样呈现客户 SMP 的存在，这样客户机操作系统才不会认为其运行在单一处理器上，才会试图初始化其他的虚拟处理器，并在其上运行调度程序。VMM 可以是模拟一个现实中的接口，例如通过 ACPI 表来表述；也可以是自定义一个接口协议，只要客户机操作系统被修改来配合 VMM 即可。

其次，我们知道，SMP 的并发执行能力虽带来了性能上的提升，但多个处理器竞争共享资源的情况也给软件实现带来了更多复杂性。为了保证 SMP 情况下多处理器访问共享资源的正确性，通常系统程序需要实现一套同步机制来协调处理器之间的步调，从而确保任何时候只有一个处理器能对共享资源进行修改，并且在释放修改权之前，确保修改的效果能够被每个处理器觉察到。在客户 SMP 机制引入后，实际上 VMM 面临着物理处理器之间（也被称为主机 SMP）以及虚拟处理器之间（即客户 SMP）的同步问题。

（1）对于发生在 VMM 自身代码之间的同步问题，由 VMM 负责协调物理处理器之间

的步调来满足主机 SMP 的要求。

(2) 对于发生在同一个虚拟机内部,多个虚拟处理器间的同步问题,通常 VMM 并不需要参与,因为这是客户机操作系统自身的职责。VMM 只需要在客户机操作系统发起某种特权操作,例如刷新页表时,正确地模拟其效果即可。

(3) 对于 VMM 造成的虚拟处理器之间的同步问题,仍然需要 VMM 来负责处理。例如,VMM 可能将 N 个虚拟处理器在 $M(M > N)$个物理处理器之间进行迁移,客户机操作系统只知道自己有 N 个虚拟处理器,所以只会在这 N 个虚拟处理器的上下文内进行同步操作,但当 VMM 将这 N 个虚拟处理器迁移到 M 个物理处理器上运行时,VMM 就必须负责所有 M 个物理处理器上状态的同步。

最后,VMM 对虚拟机管理模块也必须根据客户 SMP 的存在做相应的修改。例如,挂起命令要区分挂起虚拟处理器还是挂起虚拟机,当挂起某个虚拟机就必然挂起该虚拟机内部所有指令流的执行。

下面来看一下在客户 SMP 功能被引入后初始化过程是如何模拟的。

通常,对称多处理器技术定义有标准的一套初始化过程。在没有虚拟化的环境里,BIOS 负责选取 BSP(主启动处理器)与 AP(应用处理器),把所有处理器都初始化到某种状态后,BIOS 在 BSP 上通过启动加载程序(Boot Loader)跳转至操作系统的初始化代码,同时所有的 AP 处于某种等待初始化硬件信号的状态。接下来,操作系统会在初始化到某个时刻时,发出某种初始化硬件信号给所有的 AP,并提供一段特定的启动代码,AP 在收到初始化硬件信号后,就会跳转到操作系统指定的启动代码中继续执行。通过这样一种方式,操作系统最终就成功地按自己的方式初始化了所有的处理器,最后在每个处理器上独立地运行调度程序。

那么,在虚拟环境里,客户 SMP 功能被引入后初始化过程是怎样的?注意,此时讨论的是 VMM 已经启动运行起来,而客户机操作系统正处在初始化阶段。VMM 选择第一个虚拟处理器作为 BSP,其他虚拟处理器为 AP,把所有虚拟处理器都初始化到某种状态。这里又分为两种情况:如果客户机操作系统是不能修改的,而它又期望看到虚拟处理器与物理处理器加电重设后一样的状态,VMM 就必须按照软件开发手册上对于处理器加电重设状态的描述,设置虚拟处理器的寄存器状态,包括虚拟控制寄存器和虚拟运行模式等;如果客户机操作系统可以修改,VMM 就可以使用一套自定义的协议而不必依照规范定义的那样,例如直接跳过实模式把虚拟处理器初始化为保护模式。接下来,当启动代码初始化到某个时刻时,AP 需要收到某种初始化信号被唤醒。这里还是相应地分为两种情况:如果客户机操作系统不能被修改,则 VMM 负责截获客户机操作系统发出的 INIT-SIPI-SIPI 序列,唤醒其他虚拟的 AP;如果客户机操作系统可以被修改,VMM 也可以自定义一套简单的唤醒机制。

3.3 内存虚拟化

在介绍完处理器虚拟化基本原理之后，接着来看一下内存虚拟化。首先从一个操作系统的角度，介绍其对物理内存存在的两个主要基本认识：物理地址从0开始和内存地址连续性。而内存虚拟化的产生，主要源于VMM与客户机操作系统在对物理内存的认识上存在冲突，造成了物理内存的真正拥有者——VMM，必须对客户机操作系统所访问的内存进行一定程度上的虚拟化。所以在这一节的后一部分，会进一步介绍内存虚拟化相关的一些知识与方法。读者将会看到，内存虚拟化既满足了客户机操作系统对于内存和地址空间的特定认识，也可以更好地在虚拟机之间、虚拟机与VMM之间进行隔离，防止某个虚拟机内部的活动影响到其他的虚拟机甚至是VMM本身，从而造成安全上的漏洞。

下面先分析一下没有虚拟化的环境。在这种环境里，任何一个操作系统都认为自己完全控制处理器，相应的就完全拥有了内存的所有权。所以，操作系统总是按照一台物理计算机上内存的属性和特征对其进行管理。

那么，再来重温一下第2章对于内存和地址空间的介绍。指令对于内存的访问都是通过处理器来转发的，首先处理器会将解码后的请求发送到系统总线上，然后由芯片组来负责进一步转发。为了唯一标识，处理器采用统一编址的方式将物理内存映射成为一个地址空间，即所谓的物理地址空间。平时，我们把一根根内存条插到主板上的内存插槽中，每根内存条都需要被映射到物理地址空间中某个位置。一般来说，每根内存插槽在物理地址空间的起始地址可以在主板制造时就固定下来，也可以通过某种方式由BIOS加电后自动配置。一旦内存插槽的起始地址被固定下来，这根内存条上每个字节的物理地址就相应地确定下来了。总的来看，一根根内存条形成了一个连续的物理地址空间，而且这个物理地址空间一定是从0开始的。

例如有4个内存插槽的主板，每个插槽插上256MB的内存条，如果这4条插槽的起始地址分别固定为0x00000000、0x10000000、0x20000000和0x30000000，那么在它们上面的物理内存就被映射成0x00000000——0x0FFFFFFF、0x10000000——0x1FFFFFFF、0x20000000——0x2FFFFFFF、0x30000000——0x3FFFFFFF这4段。总的来说，这4根内存条组成该系统1GB的内存，而且这1GB的内存是从0开始的连续空间，4根内存条上每个字节都会对应到一个唯一的物理地址。处理器访问任何一个字节就是通过请求一个物理地址，芯片组收到处理器发出的内存访问请求后，就会检测内部维护的物理地址空间的分配表，当发现目标地址落在0x00000000——0x3FFFFFFF范围内时，处理器就会进一步把请求转发给内存控制器。

在没有虚拟化的环境里，操作系统也会假定物理内存是从物理地址0开始的。以x86处理器上运行Linux内核为例。在x86上，Linux内核可执行文件头里定义了每个段的大小、期望在物理地址空间中被加载的位置即1MB，以及加载后执行第一条指令的地址等，这些信息在编译连接阶段就确定下来了。由于加载的位置是1MB，那么对于后面代码，其访

问的段都是基于 1MB 这个起始地址的，这也是在编译链接阶段就确定下来了的。通常，在加载内核时，启动加载程序(Boot Loader)就会通过对该文件格式的分析，将相应的段复制到期望的位置，然后跳转到内核文件指定的入口点。而系统所做的，必须保证在该指定位置存在可用内存。如果物理地址空间不是从 0 开始的，Boot Loader 将会因指定位置找不到可用内存而拒绝加载内核，即使是把内核加载到内存中了，由于内核代码在访问段时也会自身产生错误而造成整个系统的崩溃。

除此之外，现实中的操作系统基本上对内存连续性存在一定程度的依赖性，如 DMA。DMA 的目的就是允许设备绕过处理器来直接访问物理内存，从而保证 I/O 处理的高效。目前绝大多数设备都支持 DMA 功能，只是在实现上对驱动程序提出了不同的要求。如图 3-4 所示。

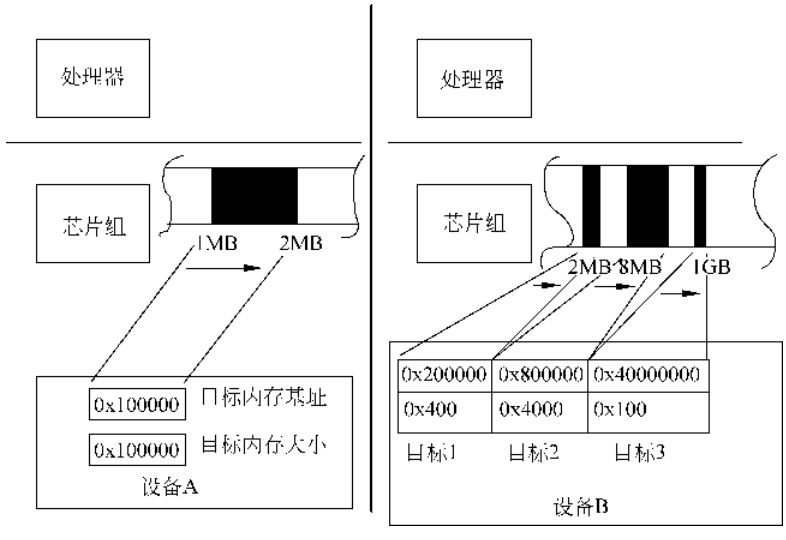

图 3-4　DMA 对内存连续性的要求

图 3-4 是一个简化的例子，现实中的设备在 DMA 的逻辑上要复杂得多。但是，这并不妨碍我们以这个简化的模型来说明问题。左边的设备使用了一种最直接的方式，即驱动程序提供 DMA 的目标内存地址 0x100000 以及大小 1MB，然后设备顺序地访问从 0x100000 到 0x200000 的内存。很容易看到，当一个内存页面大小小于 1MB 时，就需要请求几个在物理上连续的内存页面，以满足设备顺序访问内存的需求。而右边的设备则使用了一种更加灵活的方式，叫做分散-聚合(Scatter-Gather)，它允许驱动程序一次提供多个物理上不连续的内存段，设备通过相关信息来离散地访问这些不连续的目标内存。

在实际设备中，这两种模式都非常普遍，而且即使在后一种模式中，设备允许的离散块是有限的，为了支持更大的 DMA 区域，驱动程序仍然会在每一个离散块中分配多个连续的内存页面，这就意味着驱动程序必须能够从操作系统中分配到足够多连续的空闲内存页来