附录 B　虚拟机监视器

B.1　简介

多年前，IBM 将昂贵的大型机出售给大型组织，出现了一个问题：如果组织希望同时在机器上运行不同的操作系统，该怎么办？有些应用程序是在一个操作系统上开发的，有些是在其他操作系统上开发的，因此出现了该问题。作为一种解决方案，IBM 以虚拟机监视器（Virtual Machine Monitor，VMM）（也称为管理程序，hypervisor）[G74]的形式，引入了另一个间接层。

具体来说，监视器位于一个或多个操作系统和硬件之间，并为每个运行的操作系统提供控制机器的假象。然而，在幕后，实际上是监视器在控制硬件，并必须在机器的物理资源上为运行的 OS 提供多路复用。实际上，VMM 作为操作系统的操作系统，但在低得多层次上。操作系统仍然认为它与物理硬件交互。因此，透明度（transparency）是 VMM 的主要目标。

因此，我们发现自己处于一个有趣的位置：到目前为止操作系统已经成为假象提供大师，欺骗毫无怀疑的应用程序，让它们认为拥有自己私有的 CPU 和大型虚拟内存，同时在应用程序之间进行切换，并共享内存。现在，我们必须再次这样做，但这次是在操作系统之下，它曾经拥有控制权。VMM 如何为每个运行在其上的操作系统创建这种假象？

关键问题：如何在操作系统之下虚拟化机器

虚拟机监视器必须透明地虚拟化操作系统下的机器。这样做需要什么技术？

B.2　动机：为何用 VMM

今天，由于多种原因，VMM 再次流行起来。服务器合并就是一个原因。在许多设置中，人们在运行不同操作系统（甚至 OS 版本）的不同机器上运行服务，但每台机器的利用率都不高。在这种情况下，虚拟化使管理员能够将多个操作系统合并（consolidate）到更少的硬件平台上，从而降低成本并简化管理。

虚拟化在桌面上也变得流行，因为许多用户希望运行一个操作系统（比如 Linux 或 macOS X），但仍然可以访问不同平台上的本机应用程序（比如 Windows）。这种功能（functionality）上的改进也是一个很好的理由。

另一个原因是测试和调试。当开发者在一个主平台上编写代码时，他们通常希望在许多不同平台上进行调试和测试。在实际环境中，他们要将软件部署到这些平台上。因此，通过让开发人员能够在一台计算机上运行多种操作系统类型和版本，虚拟化可以轻松实现这一点。

虚拟化的复兴始于 20 世纪 90 年代中后期，由 Mendel Rosenblum 教授领导的斯坦福大学的一组研究人员推动。他的团队在用于 MIPS 处理器的虚拟机监视器 Disco [B+97]上的工作是早期的努力，它使 VMM 重新焕发活力，并最终使该团队成为 VMware [V98]的创始人，该公司现在是虚拟化技术的市场领导者。在本章中，我们将讨论 Disco 的主要技术，并尝试通过该窗口来了解虚拟化的工作原理。

B.3 虚拟化 CPU

为了在虚拟机监视器上运行虚拟机（virtual machine，即 OS 及其应用程序），使用的基本技术是受限直接执行（limited direct execution），这是我们在讨论操作系统如何虚拟化 CPU 时看到的技术。因此，如果想在 VMM 之上“启动”新操作系统，只需跳转到第一条指令的地址，并让操作系统开始运行，就这么简单。

假设我们在单个处理器上运行，并且希望在两个虚拟机之间进行多路复用，即在两个操作系统和它们各自的应用程序之间进行多路复用。非常类似于操作系统在运行进程之间切换的方式（上下文切换，context switch），虚拟机监视器必须在运行的虚拟机之间执行机器切换（machine switch）。因此，当执行这样的切换时，VMM 必须保存一个 OS 的整个机器状态（包括寄存器，PC，并且与上下文切换不同，包括所有特权硬件状态），恢复待运行虚拟机的机器状态，然后跳转到待运行虚拟机的 PC，完成切换。注意，待运行 VM 的 PC 可能在 OS 本身内（系统正在执行系统调用），或可能就在该 OS 上运行的进程内（用户模式应用程序）。

当正在运行的应用程序或操作系统尝试执行某种特权操作（privileged operation）时，我们会遇到一些稍微棘手的问题。例如，在具有软件管理的 TLB 的系统上，操作系统将使用特殊的特权指令，用一个地址转换来更新 TLB，再重新执行遇到 TLB 未命中的指令。在虚拟化环境中，不允许操作系统执行特权指令，因为它控制机器而不是其下的 VMM。因此，VMM 必须以某种方式拦截执行特权操作的尝试，从而保持对机器的控制。

如果在给定 OS 上的运行进程尝试进行系统调用，会出现 VMM 必须如何介入某些操作的简单场景。例如，进程可能尝试对一个文件调用 open()，或者可能调用 read()，从中获取数据，或者可能正在调用 fork()来创建新进程。在没有虚拟化的系统中，通过特殊指令实现系统调用。在 MIPS 上，它是一个陷阱（trap）指令。在 x86 上，它是带有参数 0x80 的 int（中断）指令。下面是 FreeBSD 上的 open 库调用[B00]（回想一下，你的 C 代码首先对 C 库进行库调用，然后执行正确的汇编序列，实际发出陷阱指令并进行系统调用）：

```
open:
    push    dword mode
    push    dword flags
    push    dword path
```

```
mov     eax, 5
push    eax
int     80h
```

在基于 UNIX 的系统上，open()只接受 3 个参数：int open(char * path, int flags, mode_t mode)。你可以在上面的代码中看到 open()库调用是如何实现的：首先，将数据推入栈（模式，标志，路径），然后将 5 推入栈，然后调用 int 80h，它将控制权转移到内核。如果你想知道，5 是用户模式应用程序与 FreeBSD 中 open()系统调用的内核之间预先商定的约定。不同的系统调用会在调用陷阱指令 int 之前将不同的数字放在栈上（在相同的位置），从而进行系统调用[①]。

执行陷阱指令时，正如之前讨论的那样，它通常会做很多有趣的事情。在我们的示例中，最重要的是它首先将控制转移（即更改 PC）到操作系统内定义良好的陷阱处理程序（trap handler）。操作系统开始启动时，会利用硬件（也是特权操作）建立此类例程的地址，因此在后续的陷阱中，硬件知道从哪里开始运行代码来处理陷阱。在陷阱的同时，硬件还做了另一件至关重要的事情：它将处理器的模式从用户模式（user mode）更改为内核模式（kernel mode）。在用户模式下，操作受到限制，尝试执行特权操作将导致陷阱，并可能终止违规进程。另一方面，在内核模式下，机器的全部能力都可用，因此可以执行所有特权操作。因此，在传统设置中（同样，没有虚拟化），控制流程如表 B.1 所示。

表 B.1　　执行系统调用

进程	硬件	操作系统
1．执行指令 （add, load, 等） 2．系统调用： 陷入 OS		
	3．切换到内核模式 跳转到陷阱处理程序	
		4．在内核模式 处理系统调用 从陷阱返回
	5．切换到用户模式 返回用户模式	
6．继续执行 （@陷阱之后的 PC）		

在虚拟化平台上，事情会更有趣。如果在 OS 上运行的应用程序希望执行系统调用，它会执行完全相同的操作：执行陷阱指令，并将参数小心地放在栈上（或寄存器中）。但是，VMM 控制机器，因此安装了陷阱处理程序的 VMM 将首先在内核模式下执行。

那么 VMM 应该如何处理这个系统调用呢？VMM 并不真正知道如何（how）处理调用。毕竟，它不知道正在运行的每个操作系统的细节，因此不知道每个调用应该做什么。然而，VMM 知道的是 OS 的陷阱处理程序在哪里（where）。它知道这一点，因为当操作系统启动时，它试图安装自己的陷阱处理程序。当操作系统这样做时，它试图执行一些特权操作，因此陷

① 使用术语“中断”来表示几乎任何理智的人都会称之为陷阱的指令，这让事情变得混乱。正如 Patterson 曾说英特尔指令集是“只有母亲才爱的 ISA。”但实际上，我们有点喜欢它，但我们不是它的母亲。

入 VMM 中。那时，VMM 记录了必要的信息（即这个 OS 的陷阱处理程序在内存中的位置）。现在，当 VMM 从在给定操作系统上运行的用户进程接收到陷阱时，它确切地知道该做什么：它跳转到操作系统的陷阱处理程序，并让操作系统按原样处理系统调用。当操作系统完成时，它会执行某种特权指令从陷阱返回（在 MIPS 上是 rett，在 x86 上是 iret），然后再次弹回 VMM，然后 VMM 意识到操作系统正试图从陷阱返回，从而执行一次真正的从陷阱返回，从而将控制返回给用户，并让机器返回用户模式。表 B.2 和表 B.3 描述了整个过程，无论是没有虚拟化的正常情况，还是虚拟化的情况（上面省略了具体的硬件操作，以节省空间）。

表 B.2 没有虚拟化的系统调用流程

进程	操作系统
1．系统调用： 陷入 OS	
	2．OS 陷阱处理程序： 解码陷阱并执行相应的系统调用例程； 完成后，从陷阱返回
3．继续执行（@陷阱之后的 PC）	

表 B.3 有虚拟化的系统调用流程

进程	操作系统	VMM
1．系统调用： 陷入 OS		
		2．进程陷入： 调用 OS 陷阱处理程序 （以减少的特权）
	3．OS 陷阱处理程序： 解码陷阱并执行系统调用 完成后：发出从陷阱返回	
		4．OS 尝试从陷阱返回： 真正从陷阱返回
5．继续执行（@陷阱之后的 PC）		

从表中可以看出，虚拟化时必须做更多的工作。当然，由于额外的跳转，虚拟化可能确实会减慢系统调用，从而可能影响性能。

你可能还注意到，我们还有一个问题：操作系统应该运行在什么模式？它无法在内核模式下运行，因为这可以无限制地访问硬件。因此，它必须以比以前更少的特权模式运行，能够访问自己的数据结构，同时阻止从用户进程访问其数据结构。

在 Disco 的工作中，Rosenblum 及其同事利用 MIPS 硬件提供的特殊模式（称为管理员模式），非常巧妙地处理了这个问题。在此模式下运行时，仍然无法访问特权指令，但可以访问比在用户模式下更多的内存。操作系统可以将这个额外的内存用于其数据结构，一切都很好。在没有这种模式的硬件上，必须以用户模式运行 OS 并使用内存保护（页表和 TLB），来适当地保护 OS 的数据结构。换句话说，当切换到 OS 时，监视器必须通过页表保护，让 OS 数据

结构的内存对 OS 可用。当切换回正在运行的应用程序时，必须删除读取和写入内核的能力。

B.4 虚拟化内存

你现在应该对处理器的虚拟化方式有了基本的了解：VMM 就像一个操作系统，安排不同的虚拟机运行。当特权级别发生变化时，会发生一些有趣的交互。但我们忽略了很大一部分：VMM 如何虚拟化内存？

每个操作系统通常将物理内存视为一个线性的页面数组，并将每个页面分配给自己或用户进程。当然，操作系统本身已经为其运行的进程虚拟化了内存，因此每个进程都有自己的私有地址空间的假象。现在我们必须添加另一层虚拟化，以便多个操作系统可以共享机器的实际物理内存，我们必须透明地这样做。

这个额外的虚拟化层使“物理”内存成为一个虚拟化层，在 VMM 所谓的机器内存（machine memory）之上，机器内存是系统的真实物理内存。因此，我们现在有一个额外的间接层：每个操作系统通过其每个进程的页表映射虚拟到物理地址，VMM 通过它的每个 OS 页面表，将生成的物理地址映射到底层机器地址。图 B.1 描述了这种额外的间接层。

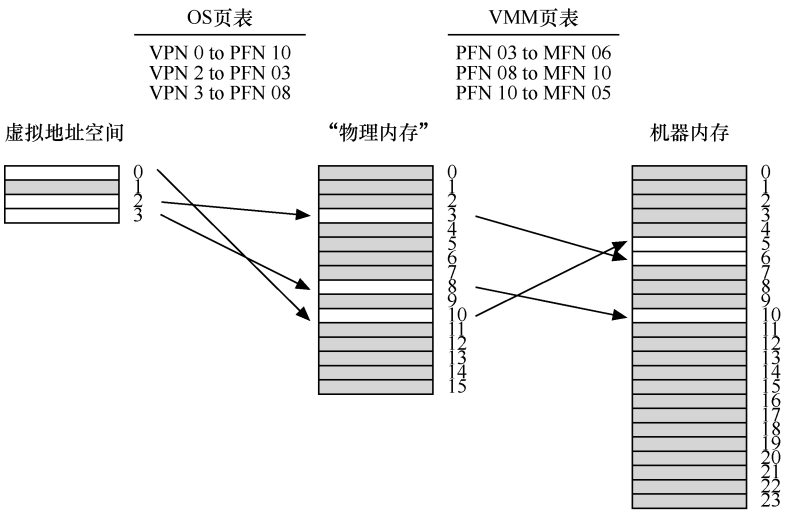

图 B.1 VMM 内存虚拟化

在该图中，只有一个虚拟地址空间，包含 4 个页面，其中 3 个是有效的（0、2 和 3）。操作系统使用其页面表将这些页面映射到 3 个底层物理帧（分别为 10、3 和 8）。在 OS 之下，VMM 提供进一步的间接级别，将 PFN 3、8 和 10 分别映射到机器帧 6、10 和 5。当然，这张图简化了一些事情。在真实系统上，会运行 V 个操作系统（V 可能大于 1），因有此 V 个 VMM 页表。此外，在每个运行的操作系统 OSi 之上，将有许多进程 Pi 运行（Pi 可能是数十或数百），因此 OSi 内有 Pi 个（每进程）页表。

为了理解它如何更好地工作，让我们回想一下地址转换（address translation）在现代分页系统中的工作原理。具体来说，让我们讨论在具有软件管理的 TLB 的系统上，在地址转

换期间发生的情况。假设用户进程生成一个地址（用于指令获取或显式加载或存储）。根据定义，该进程生成虚拟地址（virtual address），因为其地址空间已由 OS 虚拟化。如你所知，操作系统的作用是在硬件的帮助下将其转换为物理地址（virtual address），从而能够从物理内存中获取所需的内容。

假设有一个 32 位的虚拟地址空间和 4-KB 的页面大小。因此，32 位地址被分成两部分：一个 20 位的虚拟页号（VPN）和一个 12 位的偏移量。在硬件 TLB 的帮助下，OS 的作用是将 VPN 转换为有效的物理页帧号（PFN），从而产生完全形式的物理地址，可以将其发送到物理内存以获取正确的数据。在通常情况下，我们希望 TLB 能够处理硬件中的转换，从而快速实现转换。当 TLB 未命中时（至少在具有软件管理的 TLB 的系统上），操作系统必须参与处理未命中，如表 B.4 所示。

表 B.4　　没有虚拟化的 TLB 未命中流程

进程	操作系统
1．从内存加载： TLB 未命中：陷阱	
	2．OS TLB 未命中处理程序： 从 VA 提取 VPN 查找页表 如果存在并有效，则取得 PFN，更新 TLB； 从陷阱返回
3．继续执行（@导致陷入的指令的 PC） 指令重试 导致 TLB 命中	

如你所见，TLB 未命中会导致陷入操作系统，操作系统在页表中查找 VPN，将转换映射装入 TLB，来处理该故障。

然而，操作系统之下有虚拟机监视器时，事情变得更有趣了。我们再来看看 TLB 未命中的流程（参见表 B.5 的总结）。当进程进行虚拟内存引用，并导致 TLB 未命中时，运行的不是 OS TLB 的未命中处理程序。实际上，运行的是 VMM TLB 未命中处理程序，因为 VMM 是机器的真正特权所有者。但是，在正常情况下，VMM TLB 处理程序不知道如何处理 TLB 未命中，因此它立即跳转到 OS TLB 未命中处理程序。VMM 知道此处理程序的位置，因为操作系统在“启动”期间尝试安装自己的陷阱处理程序。然后运行 OS TLB 未命中处理程序，对有问题的 VPN 执行页表查找，并尝试在 TLB 中安装 VPN 到 PFN 映射。但是，这样做是一种特权操作，因此导致另一次陷入 VMM（当任何非特权代码尝试执行特权时，VMM 都会得到通知）。此时，VMM 玩了花样：VMM 不是安装操作系统的 VPN-to-PFN 映射，而是安装其所需的 VPN-to-MFN 映射。这样做之后，系统最终返回到用户级代码，该代码重试该指令，并导致 TLB 命中，从数据所在的机器帧中获取数据。

表 B.5　　有虚拟化的 TLB 未命中流程

进程	操作系统	虚拟机监视器
1．从内存加载 TLB 未命中：陷阱		

续表

进程	操作系统	虚拟机监视器
		2．VMM TLB 未命中处理程序： 调用 OS TLB 处理程序（减少的特权）
	3．OS TLB 未命中处理程序： 从 VA 提取 VPN 查找页表 如果存在并有效：取得 PFN，更新 TLB	
		4．陷阱处理程序： 非特权代码尝试更新 TLB OS 在尝试安装 VPN 到 PFN 的映射 用 VPN-to-MFN 更新 TLB（特权操作） 跳回 OS（减少的特权）
	5．从陷阱返回	
		6．陷阱处理程序： 非特权指令尝试从陷阱返回 从陷阱返回
7．继续执行（@导致陷入的指令的 PC） 指令重试 导致 TLB 命中		

这组操作还暗示了，对于每个正在运行的操作系统的物理内存，VMM 必须如何管理虚拟化。就像操作系统有每个进程的页表一样，VMM 必须跟踪它运行的每个虚拟机的物理到机器映射。在 VMM TLB 未命中处理程序中，需要查阅每个机器的页表，以便确定特定“物理”页面映射到哪个机器页面，甚至它当前是否存在于机器内存中（例如，VMM 可能已将其交换到磁盘）。

补充：管理程序和硬件管理的 TLBS

我们的讨论集中在软件管理的 TLB 以及发生未命中时需要完成的工作。但你可能想知道：有硬件管理的 TLB 时，虚拟机监视器如何参与？在这些系统中，硬件在每个 TLB 未命中时遍历页表并根据需要更新 TLB，因此 VMM 没有机会在每个 TLB 未命中时运行以将其转换到系统中。作为替代，VMM 必须密切监视操作系统对每个页表的更改（在硬件管理的系统中，由某种类型的页表基址寄存器指向），并保留一个影子页表（shadow page table），它将每个进程的虚拟地址映射到 VMM 期望的机器页面[AA06]。每当操作系统尝试安装进程的操作系统级页表时，VMM 就会安装进程的影子页表，然后硬件干活，利用影子表将虚拟地址转换为机器地址，而操作系统甚至没有注意到。

最后，你可能注意到，在这一系列操作中，虚拟化系统上的 TLB 未命中变得比非虚拟化系统更昂贵一点。为了降低这一成本，Disco 的设计人员增加了一个 VMM 级别的“软件 TLB”。这种数据结构背后的想法很简单。VMM 记录它看到操作系统尝试安装的每个虚拟到物理的映射。然后，在 TLB 未命中时，VMM 首先查询其软件 TLB 以查看它是否已经看

到此虚拟到物理映射，以及 VMM 所需的虚拟到机器的映射应该是什么。如果 VMM 在其软件 TLB 中找到转换，就将虚拟到机器的映射直接装入硬件 TLB 中，因此跳过了上面控制流中的所有来回[B+97]。

B.5 信息沟

操作系统不太了解应用程序的真正需求，因此通常必须制定通用的策略，希望对所有程序都有效。类似地，VMM 通常不太了解操作系统正在做什么或想要什么，这种知识缺乏有时被称为 VMM 和 OS 之间的信息沟（information gap），可能导致各种低效率[B+97]。例如，当 OS 没有其他任何东西可以运行时，它有时会进入空循环（idle loop），只是自旋并等待下一个中断发生：

```
while (1)
  ; // the idle loop
```

如果操作系统负责整个机器，因此知道没有其他任务需要运行，这样旋转是有意义的。但是，如果 VMM 在两个不同的操作系统下运行，一个在空循环中，另一个在运行有用的用户进程，那么 VMM 知道一个操作系统处于空闲状态会很有用，这样可以为做有用工作的操作系统提供更多的 CPU 时间。

补充：半虚拟化

在许多情况下，最好是假定，无法为了更好地使用虚拟机监视器而修改操作系统（例如，因为你在不友好的竞争对手的操作系统下运行 VMM）。但是，情况并非总是如此。如果可以修改操作系统（正如我们在页面按需置零的示例中所见），它可能在 VMM 上更高效地运行。运行修改后的操作系统，以便在 VMM 上运行，这通常称为半虚拟化（para-virtualization）[WSG02]，因为 VMM 提供的虚拟化不是完整的虚拟化，而是需要操作系统更改才能有效运行的部分虚拟化。研究表明，一个设计合理的半虚拟化系统，只需要正确的操作系统更改，就可以接近没有 VMM 时的效率[BD+03]。

另一个例子是页面按需置零。大多数操作系统在将物理帧映射到进程的地址空间之前将其置零。这样做的原因很简单：安全性。如果操作系统为一个进程提供了另一个已经使用的页面，但没有将其置零，则可能会发生跨进程的信息泄露，从而可能泄露敏感信息。遗憾的是，出于同样的原因，VMM 必须将它提供给每个操作系统的页面置零，因此很多时候页面将置零两次，一次由 VMM 分配给操作系统，一次由操作系统分配给操作系统的一个进程。Disco 的作者没有很好地解决这个问题的方法：他们只是简单地将操作系统（IRIX）改为不对页面置零，因为知道已被底层 VMM [B+97]置零。

类似这样的问题，这里描述的还有很多。一种解决方案是 VMM 使用推理（一种隐含信息，implicit information）来克服该问题。例如，VMM 可以通过注意到 OS 切换到低功率模式来检测空闲循环。在半虚拟化（para-virtualized）系统中，还有另一种方法，需要更改操作系统。这种更明确的方法虽然难以实施，但却非常有效。

B.6 小结

虚拟化正在复兴。出于多种原因，用户和管理员希望同时在同一台计算机上运行多个操作系统。关键是 VMM 通常透明地（transparently）提供服务，上面的操作系统完全不知道它实际上并没有控制机器的硬件。VMM 使用的关键方法是扩展受限直接执行的概念。通过设置硬件，让 VMM 能够介入关键事件（例如陷阱），VMM 可以完全控制机器资源的分配方式，同时保留操作系统所需的假象。

提示：使用隐含信息

隐含信息可以成为分层系统中的一个强大工具，在这种系统中很难改变系统之间的接口，但需要更多关于系统不同层的信息。例如，基于块的磁盘设备，可能想了解更多关于它上面的文件系统如何使用它的信息。同样，应用程序可能想知道文件系统页面缓存中当前有哪些页面，但操作系统不提供访问此信息的 API。在这两种情况下，研究人员都开发了强大的推理技术，来隐式收集所需的信息，而无需在层[AD+01，S+03]之间建立明确的接口。这些技术在虚拟机监视器中非常有用，它希望了解有关在其上运行的 OS 的更多信息，而无需在两个层之间使用显式 API。

你可能已经注意到，操作系统为进程执行的操作与 VMM 为操作系统执行的操作之间存在一些相似之处。它们毕竟都是虚拟化硬件，因此做了一些相同的事情。但是，有一个关键的区别：通过操作系统虚拟化，提供了许多新的抽象和漂亮的接口。使用 VMM 级虚拟化，抽象与硬件相同（因此不是很好）。虽然 OS 和 VMM 都虚拟化硬件，但它们通过提供完全不同的接口来实现。与操作系统不同，VMM 没有特别打算让硬件更易于使用。

如果你想了解有关虚拟化的更多信息，还有许多其他主题需要研究。例如，我们甚至没有讨论 I/O 会发生什么，这个主题在虚拟化平台方面有一些有趣的新问题。我们也没有讨论操作系统“作为兼职”运行在有时称为“托管”配置中，虚拟化如何工作。如果你感兴趣，请阅读有关这两个主题的更多信息[SVL01]。我们也没有讨论，如果 VMM 上运行的一些操作系统占用太多内存，会发生什么。

最后，硬件支持改变了平台支持虚拟化的方式。英特尔和 AMD 等公司现在直接支持额外的虚拟化层，从而避免了本章中的许多软件技术。也许，在尚未撰写的一章中，我们会更详细地讨论这些机制。

参考资料

[AA06]“A Comparison of Software and Hardware Techniques for x86 Virtualization”

Keith Adams and Ole Agesen

ASPLOS ’06, San Jose, California

来自两位 VMware 工程师的一篇优秀的论文，讲述了为虚拟化提供硬件支持所带来的惊人的小优势。此外，还有关于 VMware 虚拟化的一般性讨论，包括为了虚拟化难以虚拟化的 x86 平台，而必须采用的疯狂的二

进制翻译技巧。

[AD+01]“Information and Control in Gray-box Systems”Andrea C. Arpaci-Dusseau and Remzi H. Arpaci-Dusseau SOSP ’01, Banff, Canada
我们自己的工作是如何推断信息，甚至从应用程序级别对操作系统施加控制，而不对操作系统进行任何更改。其中最好的例子：使用基于概率探测器的技术确定在 OS 中缓存哪些文件块。这样做可以让应用程序更好地利用缓存，优先安排会导致命中的工作。

[B00]“FreeBSD Developers’ Handbook:
Chapter 11 x86 Assembly Language Programming”
一本 BSD 开发者手册中关于系统调用的很好的教程。

[BD+03]“Xen and the Art of Virtualization”
Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neuge- bauer, Ian Pratt, Andrew Warfield
SOSP ’03, Bolton Landing, New York
该论文表明，对于半虚拟化系统，虚拟化系统的开销可以低得令人难以置信。这篇关于 Xen 虚拟机监视器的论文如此成功，导致了一家公司的诞生。

[B+97]“Disco: Running Commodity Operating Systems on Scalable Multiprocessors”
Edouard Bugnion, Scott Devine, Kinshuk Govil, Mendel Rosenblum
SOSP ’97
将系统社区重新带回虚拟机研究的论文。好吧，也许这是不公平的，因为 Bressoud 和 Schneider [BS95]也做了，但在这里我们开始理解为什么虚拟化会回来。然而更令人瞩目的是，这群优秀的研究人员创立了 VMware，赚取了数十亿美元。

[BS95]“Hypervisor-based Fault-tolerance”Thomas C. Bressoud, Fred B. Schneider SOSP ’95
最早引入虚拟机管理程序（hypervisor，这只是虚拟机监视器的另一个术语）的论文之一。然而，在这项工作中，这些管理程序用于提高硬件故障的系统容忍度，这可能不如本章讨论的一些更实际的场景有用。但它本身仍然是一篇非常有趣的论文。

[G74]“Survey of Virtual Machine Research”
R.P. Goldberg
IEEE Computer, Volume 7, Number 6
一份对许多老的虚拟机研究的调查。

[SVL01]“Virtualizing I/O Devices on VMware Workstation’s Hosted Virtual Machine Monitor”
Jeremy Sugerman, Ganesh Venkitachalam and Beng-Hong Lim
USENIX ’01, Boston, Massachusetts
本文很好地概述了在使用托管体系结构的 VMware 中 I/O 的工作方式。该体系结构利用了许多操作系统自身的功能，避免了在 VMM 中重新实现它们。

[V98] VMware corporation.
这可能是本书中最无价值的参考资料，因为你可以自己阅读一下。但无论如何，该公司成立于 1998 年，是虚拟化领域的领导者。

[S+03]“Semantically-Smart Disk Systems”
Muthian Sivathanu, Vijayan Prabhakaran, Florentina I. Popovici, Timothy E. Denehy, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau FAST ’03, San Francisco, California, March 2003
又是我们的工作，这次展示了一个基于块的设备如何能够推断出它上面的文件系统正在做什么，例如删除文件。其中使用的技术在块设备内实现了有趣的新功能，例如安全删除或更可靠的存储。

[WSG02]“Scale and Performance in the Denali Isolation Kernel”Andrew Whitaker, Marianne Shaw, and Steven D. Gribble
OSDI ’02, Boston, Massachusetts
介绍术语半虚拟化的论文。虽然人们可以争辩说 Bugnion 等人[B+97]在 Disco 论文中介绍了半虚拟化的概念，但 Whitaker 等人进一步说明，这个想法的通用性如何超出以前的想象。