本是否仍然有效。如果是这样，C1 可以使用它。如果不是，C1 应该从服务器获取更新的版本。

崩溃后服务器恢复也更复杂。问题是回调被保存在内存中。因此，当服务器重新启动时，它不知道哪个客户端机器具有哪些文件。因此，在服务器重新启动时，服务器的每个客户端必须意识到服务器已崩溃，并将其所有缓存内容视为可疑，并且（如上所述）在使用之前重新检查文件的有效性。因此，服务器崩溃是一件大事，因为必须确保每个客户端及时了解崩溃，或者冒着客户端访问陈旧文件的风险。有很多方法可以实现这种恢复。例如，让服务器在每个客户端启动并再次运行时向每个客户端发送消息（说“不要信任你的缓存内容!”），或让客户端定期检查服务器是否处于活动状态（利用心跳（heartbeat）消息，正如其名）。如你所见，构建更具可扩展性和合理性的缓存模型需要付出代价。使用 NFS，客户端很少注意到服务器崩溃。

49.7 AFSv2 的扩展性和性能

有了新协议，人们对 AFSv2 进行了测量，发现它比原来的版本更具可扩展性。实际上，每台服务器可以支持大约 50 个客户端（而不是仅仅 20 个）。另一个好处是客户端性能通常非常接近本地性能，因为在通常情况下，所有文件访问都是本地的。文件读取通常转到本地磁盘缓存（可能还有本地内存）。只有当客户端创建新文件或写入现有文件时，才需要向服务器发送 Store 消息，从而用新内容更新文件。

对于常见的文件系统访问场景，通过与 NFS 进行比较，可以对 AFS 的性能有所了解。表 49.4 展示了定性比较的结果。

表 49.4 比较：AFS 与 NFS

工作负载	NFS	AFS	AFS/NFS
1．小文件，顺序读取	$N_s \cdot$ Lnet	$N_s \cdot$ Lnet	1
2．小文件，顺序重读	$N_s \cdot$ Lmem	$N_s \cdot$ Lmem	1
3．中文件，顺序读取	$N_m \cdot$ Lnet	$N_m \cdot$ Lnet	1
4．中文件，顺序重读	$N_m \cdot$ Lmem	$N_m \cdot$ Lmem	1
5．大文件，顺序读取	$N_L \cdot$ Lnet	$N_L \cdot$ Lnet	1
6．大文件，顺序重读	$N_L \cdot$ Lnet	$N_L \cdot$ Ldisk	Ldisk / Lnet
7．大文件，单次读取	Lnet	$N_L \cdot$ Lnet	N_L
8．小文件，顺序写入	$N_s \cdot$ Lnet	$N_s \cdot$ Lnet	1
9．大文件，顺序写入	$N_L \cdot$ Lnet	$N_L \cdot$ Lnet	1
10．大文件，顺序覆写	$N_L \cdot$ Lnet	$2 \cdot N_L \cdot$ Lnet	2
11．大文件，单次写入	Lnet	$2 \cdot N_L \cdot$ Lnet	$2 \cdot N_L$

在表中，我们分析了不同大小的文件的典型读写模式。小文件中有 N_S 块，中等文件有 N_M 个块，大文件有 N_L 块。假设中小型文件可以放入客户端的内存，大文件可以放入本地磁

盘，但不能放入客户端内存。

为了便于分析，我们还假设，跨网络访问远程服务器上的文件块，需要的时间为 L_{net}。访问本地内存需要 L_{mem}，访问本地磁盘需要 L_{disk}。一般假设是 $L_{net} > L_{disk} > L_{mem}$。

最后，我们假设第一次访问文件没有任何缓存命中。如果相关高速缓存具有足够容量来保存文件，则假设后续文件访问（即“重新读取”）将在高速缓存中命中。

该表的列展示了特定操作（例如，小文件顺序读取）在 NFS 或 AFS 上的大致时间。最右侧的列展示了 AFS 与 NFS 的比值。

我们有以下观察结果。首先，在许多情况下，每个系统的性能大致相当。例如，首次读取文件时（即工作负载 1、3、5），从远程服务器获取文件的时间占主要部分，并且两个系统上差不多。在这种情况下，你可能会认为 AFS 会更慢，因为它必须将文件写入本地磁盘。但是，这些写入由本地（客户端）文件系统缓存来缓冲，因此上述成本可能不明显。同样，你可能认为从本地缓存副本读取 AFS 会更慢，因为 AFS 会将缓存副本存储在磁盘上。但是，AFS 再次受益于本地文件系统缓存。读取 AFS 可能会命中客户端内存缓存，性能与 NFS 类似。

其次，在大文件顺序重新读取时（工作负载 6），出现了有趣的差异。由于 AFS 具有大型本地磁盘缓存，因此当再次访问该文件时，它将从磁盘缓存中访问该文件。相反，NFS 只能在客户端内存中缓存块。结果，如果重新读取大文件（即比本地内存大的文件），则 NFS 客户端将不得不从远程服务器重新获取整个文件。因此，假设远程访问确实比本地磁盘慢，AFS 在这种情况下比 NFS 快一倍。我们还注意到，在这种情况下，NFS 会增加服务器负载，这也会对扩展产生影响。

第三，我们注意到，（新文件的）顺序写入应该在两个系统上性能差不多（工作负载 8、9）。在这种情况下，AFS 会将文件写入本地缓存副本。当文件关闭时，AFS 客户端将根据协议强制写入服务器。NFS 将缓冲写入客户端内存，可能由于客户端内存压力，会强制将某些块写入服务器，但在文件关闭时肯定会将它们写入服务器，以保持 NFS 的关闭时刷新的一致性。你可能认为 AFS 在这里会变慢，因为它会将所有数据写入本地磁盘。但是，要意识到它正在写入本地文件系统。这些写入首先提交到页面缓存，并且只是稍后（在后台）提交到磁盘，因此 AFS 利用了客户端操作系统内存缓存基础结构的优势，提高了性能。

第四，我们注意到 AFS 在顺序文件覆盖（工作负载 10）上表现较差。之前，我们假设写入的工作负载也会创建一个新文件。在这种情况下，文件已存在，然后被覆盖。对于 AFS 来说，覆盖可能是一个特别糟糕的情况，因为客户端先完整地提取旧文件，只是为了后来覆盖它。相反，NFS 只会覆盖块，从而避免了初始的（无用）读取[①]。

最后，访问大型文件中的一小部分数据的工作负载，在 NFS 上比 AFS 执行得更好（工作负载 7、11）。在这些情况下，AFS 协议在文件打开时获取整个文件。遗憾的是，只进行了一次小的读写操作。更糟糕的是，如果文件被修改，整个文件将被写回服务器，从而使性能影响加倍。NFS 作为基于块的协议，执行的 I/O 与读取或写入的大小成比例。

总的来说，我们看到 NFS 和 AFS 做出了不同的假设，并且因此实现了不同的性能结果，这不意外。这些差异是否重要，总是要看工作负载。

① 我们假设 NFS 读取是按照块大小和块对齐的。如果不是，NFS 客户端也必须先读取该块。我们还假设文件未使用 O_TRUNC 标志打开。如果是，AFS 中的初始打开也不会获取即将被截断的文件内容。

49.8 AFS：其他改进

就像我们在介绍 Berkeley FFS（添加了符号链接和许多其他功能）时看到的那样，AFS 的设计人员在构建系统时借此机会添加了许多功能，使系统更易于使用和管理。例如，AFS 为客户端提供了真正的全局命名空间，从而确保所有文件在所有客户端计算机上以相同的方式命名。相比之下，NFS 允许每个客户端以它们喜欢的任何方式挂载 NFS 服务器，因此只有通过公约（以及大量的管理工作），才能让文件在不同客户端上有相似的名字。

补充：工作负载的重要性

评估任何系统都有一个挑战：选择工作负载（workload）。由于计算机系统以多种不同的方式使用，因此有多种工作负载可供选择。存储系统设计人员应该如何确定哪些工作负载很重要，以便做出合理的设计决策？

鉴于他们在测量文件系统使用方式方面的经验，AFS 的设计者做出了某些工作负载假设。具体来说，他们认为大多数文件不会经常共享，而是按顺序完整地访问。鉴于这些假设，AFS 设计非常有意义。

但是，这些假设并非总是正确。例如，想象一个应用程序，它定期将信息追加到日志中。这些小的日志写入会将少量数据添加到现有的大型文件中，这对 AFS 来说是个问题。还存在许多其他困难的工作负载，例如，事务数据库中的随机更新。

要了解什么类型的工作负载常见，一种方法是通过人们已经进行的各种研究。请参考这些研究[B+91，H+11，R+00，V99]，看看工作量分析得好例子，包括 AFS 的回顾[H+88]。

AFS 也认真对待安全性，采用了一些机制来验证用户，确保如果用户需要，可以让一组文件保持私密。相比之下，NFS 在多年里对安全性的支持非常原始。

AFS 还包含了灵活的、用户管理的访问控制功能。因此，在使用 AFS 时，用户可以很好地控制谁可以访问哪些文件。与大多数 UNIX 文件系统一样，NFS 对此类共享的支持要少得多。

最后，如前所述，AFS 添加了一些工具，让系统管理员可以更简单地管理服务器。在考虑系统管理方面，AFS 遥遥领先。

49.9 小结

AFS 告诉我们，构建分布式文件系统与我们在 NFS 中看到的完全不同。AFS 的协议设计特别重要。通过让服务器交互最少（通过全文件缓存和回调），每个服务器可以支持许多客户端，从而减少管理特定站点所需的服务器数量。许多其他功能，包括单一命名空间、安全性和访问控制列表，让 AFS 非常好用。AFS 提供的一致性模型易于理解和推断，不会导致偶尔在 NFS 中看到的奇怪行为。

也许很不幸，AFS 可能在走下坡路。由于 NFS 是一个开放标准，许多不同的供应商都

支持它，它与 CIFS（基于 Windows 的分布式文件系统协议）一起，在市场上占据了主导地位。虽然人们仍不时看到 AFS 安装（例如在各种教育机构，包括威斯康星大学），但唯一持久的影响可能来自 AFS 的想法，而不是实际的系统本身。实际上，NFSv4 现在添加了服务器状态（例如，“open”协议消息），因此与基本 AFS 协议越来越像。

参考资料

[B+91]“Measurements of a Distributed File System”
Mary Baker, John Hartman, Martin Kupfer, Ken Shirriff, John Ousterhout SOSP ’91, Pacific Grove, California, October 1991
早期的论文，测量人们如何使用分布式文件系统，符合 AFS 中的大部分直觉。

[H+11]“A File is Not a File: Understanding the I/O Behavior of Apple Desktop Applications”Tyler Harter, Chris Dragga, Michael Vaughn,
Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
SOSP ’11, New York, New York, October 2011
我们自己的论文，研究 Apple Desktop 工作负载的行为。事实证明，它们与系统研究社区通常关注的许多基于服务器的工作负载略有不同。本文也是一篇优秀的参考文献，指出了很多相关的工作。

[H+88]“Scale and Performance in a Distributed File System”
John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, Michael J. West
ACM Transactions on Computing Systems (ACM TOCS), page 51-81, Volume 6, Number 1, February 1988
著名的 AFS 系统的期刊长版本，该系统仍然在全世界的许多地方使用，也可能是关于如何构建分布式文件系统的最早的清晰思考。它是测量科学和原理工程的完美结合。

[R+00]“A Comparison of File System Workloads”Drew Roselli, Jacob R. Lorch, Thomas E. Anderson
USENIX ’00, San Diego, California, June 2000
与 Baker 的论文[B+91]相比，有最近的一些记录。

[S+85]“The ITC Distributed File System: Principles and Design”
M．Satyanarayanan, J.H. Howard, D.A. Nichols, R.N. Sidebotham, A. Spector, M.J. West SOSP ’85, Orcas Island, Washington, December 1985
关于分布式文件系统的较早的文章。AFS 的许多基本设计都在这个较早的系统中实现，但没有对扩展进行改进。

[V99]“File system usage in Windows NT 4.0”Werner Vogels
SOSP '99, Kiawah Island Resort, South Carolina, December 1999
对 Windows 工作负载的一项很酷的研究，与之前已经完成的许多基于 UNIX 的研究相比，它在本质上是不同的。

作业

本节引入了 afs.py，这是一个简单的 AFS 模拟器，可用于增强你对 Andrew 文件系统工作原理的理解。阅读 README 文件，以获取更多详细信息。

问题

1．运行一些简单的场景，以确保你可以预测客户端将读取哪些值。改变随机种子标志（-s），看看是否可以追踪并预测存储在文件中的中间值和最终值。还可以改变文件数（-f），客户端数（-C）和读取比率（-r，介于 0 到 1 之间），这样更有挑战。你可能还想生成稍长的追踪，记录更有趣的交互，例如（-n 2 或更高）。

2．现在执行相同的操作，看看是否可以预测 AFS 服务器发起的每个回调。尝试使用不同的随机种子，并确保使用高级别的详细反馈（例如，-d 3），来查看当你让程序计算答案时（使用-c），何时发生回调。你能准确猜出每次回调发生的时间吗？发生一次回调的确切条件是什么？

3．与上面类似，运行一些不同的随机种子，看看你是否可以预测每一步的确切缓存状态。用-c 和-d 7 运行，可以观察到缓存状态。

4．现在让我们构建一些特定的工作负载。用-A oa1:w1:c1,oa1:r1:c1 标志运行该模拟程序。在用随机调度程序运行时，客户端 1 在读取文件 a 时，观察到的不同可能值是什么（尝试不同的随机种子，看看不同的结果）？在两个客户端操作的所有可能的调度重叠中，有多少导致客户端 1 读取值 1，有多少读取值 0？

5．现在让我们构建一些具体的调度。当使用-A oa1:w1:c1,oa1:r1:c1 标志运行时，也用以下调度方案来运行：-S 01, -S 100011, -S 011100，以及其他你可以想到的调度方案。客户端 1 读到什么值？

6．现在使用此工作负载来运行：-A oa1:w1:c1,oa1:w1:c1，并按上述方式更改调度方式。用-S 011100 运行时会发生什么？用-S 010011 时怎么样？确定文件的最终值有什么重要意义？

第 50 章　关于分布式的总结对话

学生：嗯，真快。在我看来，真是太快了！

教授：是的，分布式系统又复杂又酷，值得学习。但不属于本书（或本课程）的范围。

学生：那太糟糕了，我想了解更多！但我确实学到了一些知识。

教授：比如？

学生：嗯，一切都会失败。

教授：好的开始。

学生：但是通过拥有大量这些东西（无论是磁盘、机器还是其他东西），可以隐藏出现的大部分失败。

教授：继续！

学生：像重试这样的一些基本技巧非常有用。

教授：确实。

学生：你必须仔细考虑协议：机器之间交换的确切数据位。协议可以影响一切，包括系统如何响应故障，以及它们的可扩展性。

教授：你真是学得越来越好。

学生：谢谢！您本人也不是差劲的老师！

教授：非常感谢。

学生：那么本书结束了吗？

教授：我不确定。他们没有给我任何通知。

学生：我也不确定。我们走吧。

教授：好的。

学生：您先请。

教授：不，你先。

学生：教授先请。

教授：不，你先，我在你之后。

学生：（被激怒）那好！

教授：（等待）……那你为什么不离开？

学生：我不知道怎么做。事实证明，我唯一能做的就是参与这些对话。

教授：我也是。现在你已经学到了我们的最后一课……

附录 A　关于虚拟机监视器的对话

学生：所以现在我们被困在附录中了，对吧？

教授：是的，就在你认为事情变得更糟的时候。

学生：嗯，我们要谈什么？

教授：一个重生的老话题：虚拟机监视器（virtual machine monitor），也称为虚拟机管理程序（hypervisor）。

学生：哦，就像 VMware 一样？这很酷，我以前用过这种软件。

教授：确实很酷。我们将了解 VMM 如何在系统中添加另一层虚拟化，这一系统在操作系统本身之下！真的，疯狂而惊人的东西。

学生：听起来很好。为什么不在本书的前面部分中包含本章，然后包含虚拟化？真的不应该放在那里吗？

教授：我想，这超过了我们的职责范围。但我猜是因为：那里已有很多材料。通过将善于 VMM 的这一小部分移到附录中，教师可以选择是包含它还是跳过它。但我确实认为它应该被包括在内，因为如果你能理解 VMM 是如何工作的，那就真的非常了解虚拟化了。

学生：好吧，让我们开始工作吧！

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。

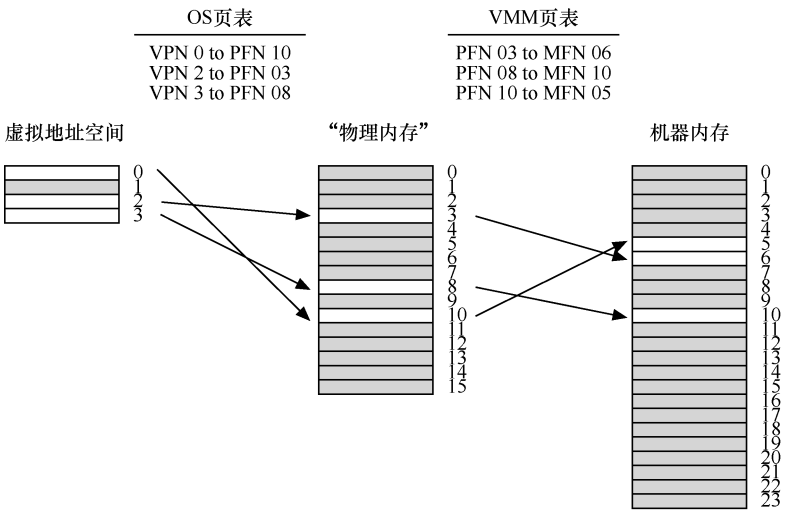

图 B.1 VMM 内存虚拟化

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。