第3章

进　程

在本章中我们将详细讨论各种进程在分布式系统中是如何发挥重要作用的。进程的概念源自于操作系统，它定义为执行中的程序。从操作系统的角度来说，进程管理和调度也许是它要处理的最重要的问题。但是对于分布式系统来说，还存在许多同等甚至更加重要的问题。

例如，为了有效地组织客户-服务器系统，通常使用多线程技术更为方便。在分布式系统中，线程的主要作用就是以适当的方式来构建客户和服务器，使得通信和本地处理过程可以并行进行，从而获得性能上的提高。

虚拟化(virtulization)是近几年来开始流行的一个概念。它允许一个应用程序(可能连同操作系统在内的所有运行环境)并行地与其他程序一同运行，但高度独立于底层硬件和**平台**(platform)，从而达到高度**可移植性**(portability)。另外，虚拟化也有利于隔离由错误和安全问题引起的系统**失效**(failure)。我们将专用一节来特别讨论这个重要的概念。

我们在第2章中曾经指出，客户-服务器组织结构在分布式系统中是很重要的。在本章中，我们将对客户和服务器的典型组织结构作详细的剖析。另外，我们还将考察常见的服务器设计问题。

进程在不同机器之间的迁移是一个重要问题，对于广域分布式系统来说，这个问题显得特别重要。进程的迁移(更明确地说是代码的迁移)有助于获得可扩展性，也可以帮助动态地配置客户和服务器。本章将讨论代码迁移的确切含义及其相关内容。

3.1 线　程

虽然**进程**(process)构成了分布式系统中的基本组成单元，但是实践表明，操作系统提供的用于构建分布式系统的进程在粒度上还是太大了。而就粒度而言，将每个进程细分为若干控制**线程**(thread)的形式则更加合适，可以使构建分布式应用程序变得更加方便，并可获得更好的性能。在本节中，我们将详细分析线程在分布式系统中所扮演的角色，并且说明线程的重要性。关于线程以及使用线程构建应用程序的更详细内容请参见(Lewis 和 Berg 1998)以及(Stevens 1999)。

3.1.1 线程简介

为了理解线程在分布式系统中所扮演的角色，重要的是要了解进程是什么，以及进程与线程之间的关系。为了程序执行的需要，操作系统创建多个虚拟处理器，每个虚拟处理器运行一个程序。为了保持对这些虚拟处理器的跟踪，操作系统中有一张**进程表**(process table)，其包含的条目中存储着 CPU 寄存器值、内存映像、打开的文件、统计信息、特权信息等。进程一般定义为执行中的程序，也就是当前在操作系统的某个虚拟处理器上运行的一个程序。操作系统特别注意确保独立的进程不会有意或者无意地破坏其他独立进程运行的正确性。也就是说，多个进程并发地共享同一个 CPU 以及其他硬件资源这样一个事实是透明的。一般说来，操作系统需要硬件支持来实现这种隔离。

要得到这种并发透明性需要付出相对较高的代价。例如，每次创建一个进程的时候，操作系统必须分配一个完整的独立地址空间。空间分配意味着要对内存段进行初始化，比如先对数据段清零，将有关的程序复制到文本段中，随后为临时数据建立堆栈等。在两个进程之间切换 CPU 的开销同样会比较大，除了要保存 CPU 环境(包括寄存器值、程序计数器、堆栈指针等)以外，操作系统还必须修改**内存管理单元**(memory management unit，MMU)的寄存器，并且将位于**转换后备缓冲器**(translation lookaside buffer，TLB)中的地址转换缓存内容标记为无效。另外，如果操作系统支持同时运行的进程数目超出主存容纳能力，则必须在切换进程之前先在主存和磁盘之间进行交换。

类似于进程的概念，一个线程独立地执行它自己的程序代码。然而，线程与进程不同的是，如果要取得高度的并发透明性会导致性能降低的话，那么线程就不会有这样的企图。因此，线程系统一般只维护用来让多个线程共享 CPU 所必需的最少量信息。特别是，**线程上下文**(thread context)中一般只包含 CPU 上下文以及某些其他的线程管理信息。比如说，线程系统可能会记录线程当前正在某个互斥变量上阻塞的事实，这样，系统就不会选择该线程执行。那些对于多线程管理不是完全必要的信息通常被忽略。由于这个原因，在单个进程中防止数据遭到某些线程不合法的访问的任务完全落在了应用程序开发人员的肩上。

这种方法有两个重要含义。首先，多线程应用程序的性能至少不会比其单线程版本差。事实上，在很多情况下，多线程能够提高性能。其次，由于线程并不像进程那样彼此隔离并受到系统自动提供的保护的，因此多线程应用程序的开发需要付出更多的努力。设计合理并实行简单的原则会大有益处。不幸的是，现在的实践却证明这条原则并没有得到很好的理解。

3.1.1.1 非分布式系统中的线程用法

在讨论线程在分布式系统中的作用之前，首先让我们来考察它们的传统用法——在非分布式系统中的用法。在进程中使用多线程有许多好处。这就使得多线程系统的使用变得流行起来。

多线程最显著的好处来自以下事实，那就是在只拥有单线程的进程中，一旦执行了造成阻塞的系统调用，整个进程就被阻塞了。为了说明这一点，我们来考虑一下诸如电子表格这样的应用程序，并且假定用户需要连续并且交互地改变值。电子表格程序的重要特点是，需

要维护不同单元格之间的函数依赖关系，而这些单元格常常位于不同的数据表中。因此，一旦修改了某个单元格，所有关联的单元格都要自动更新。如果用户对某个单元格中的值进行改动，将会触发大量计算操作。如果只有单个控制线程，在程序等待输入的时候就无法进行计算，同时也很难在进行计算的时候接受输入。最方便的解决办法是使用至少两个控制线程：其中一个管理与用户之间的交互，而另一个进行数据表的更新工作。当这两个线程在并行工作的时候，还可以用第三个线程来备份表格数据到硬盘。

多线程的另一个优点是，在多处理器系统上执行多线程程序的时候，可以使用并行操作技术。在这种情况下，每一个线程占用一个 CPU，而共享的数据则存储在共享主存中。如果设计得当，这种并行操作可以是透明的：进程可以与运行在单处理器系统上一样好，只是慢一些。随着相对便宜的多处理器工作站的出现，并行操作多线程技术变得愈发重要。这种计算机系统的典型应用是在客户-服务器应用程序中用来运行服务器程序。

多线程技术在大型应用程序上下文中也是很有用的。这种应用程序一般是作为一组相互协作的程序开发出来的，其中每一个程序都通过独立的进程执行。这种方法的典型例子是 UNIX 系统。程序间协作是通过进程间通信(IPC)机制实现的。对于 UNIX 系统来说，这套机制中通常包括已命名管道、消息队列以及共享内存段，请见(Stevens 1992)。所有 IPC 机制都有一个主要的缺陷，就是其中的通信需要开销庞大的上下文切换，如图 3.1 中三个点所示。

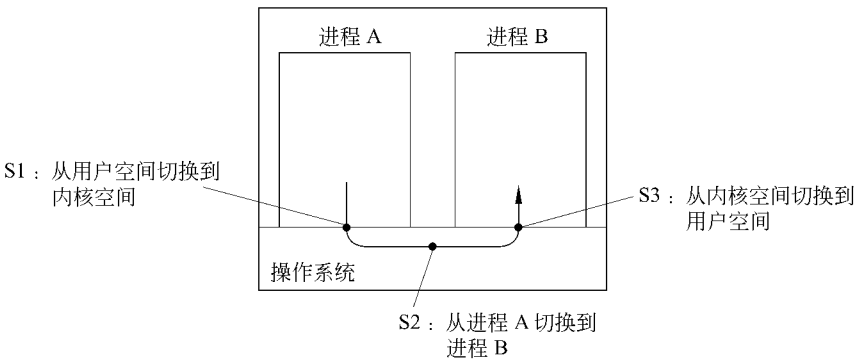

图 3.1 IPC 造成的上下文切换

由于 IPC 需要内核干预才能进行，因此，要进行 IPC 的进程一般首先要从用户模式切换到内核模式，如图 3.1 中的 S1 所示。这就需要改变 MMU 中的内存映像，同时还要刷新 TLB。在内核中进行进程上下文的切换(图 3.1 中的 S2)，随后就可以从内核模式切换回用户模式以使得通信的另一方能够激活。后一次切换也同样需要改变 MMU 映像并且刷新 TLB。

除了像上面那样使用进程来构建应用程序，我们也可以使用独立的线程来执行不同的组成部分。各个组成部分之间的通信完全通过数据共享来处理。线程切换可以完全在用户空间中完成，也可以由内核来掌管线程并对它们进行调度。这样在性能上就可以得到极大的提升。

最后，使用线程还出于纯粹的软件工程上的考虑。有许多应用程序如果用一组相互协作的线程来构建，常常是比较简便的。想象一下那些需要完成若干任务(这些任务或多或少

是独立的)的应用程序。比如说,在字处理器中,可以用独立的线程来分别处理用户输入、拼写检查及语法检查、文档布局、索引生成等工作。

3.1.1.2 线程的实现

线程一般是以线程包的形式提供的。这种包中含有创建与销毁线程的操作,以及对诸如互斥变量以及条件变量之类的同步化变量的操作。有两种实现线程包的基本方法:第一个方法是可以构造一个完全在用户模式下执行的线程库;另一个方法是由内核来掌管线程并进行调度。

采用用户级线程库有很多好处。首先,创建和销毁线程的开销很小。由于所有线程管理工作都保持在用户地址空间中进行,线程创建的开销主要取决于为线程堆栈的建立而分配内存的开销。与此类似,销毁线程的工作主要是释放线程堆栈所占用的内存,因为线程销毁后它将不会再使用这些内存。这些操作的开销都不大。

采用用户级线程的好处还有,可以通过为数不多的几条指令来实现线程上下文的切换。基本上,只有 CPU 寄存器值需要存储,并随后用将要切换到的线程的原先存储的值重新加载到 CPU 寄存器中去,并不需要内存映像的改变、TLB 的刷新以及 CPU 统计等。线程上下文的切换是在两个线程需要同步的时候发生的,比如说在进入共享数据段的时候。

然而,用户级线程的主要缺陷在于,对引起阻塞的系统调用的调用将会立即阻塞该线程所属的整个进程,也就阻塞了所属进程中的所有其他线程。我们曾经说过,如果要将大型应用程序分为若干部分来构建,所有部分在逻辑上可以同时执行,那么线程是非常有用的。在这种情况下,在 I/O 过程中发生的阻塞不应该妨碍此时其他部分的执行。对于有这种需要的应用程序,用户级线程没什么益处。

这些问题大都可以通过在操作系统的内核实现线程的方法来解决。不幸的是,这种解决方法的代价非常大:每一个线程操作(创建、删除、同步化等)都必须由内核来执行,需要进行系统调用。线程上下文的切换的开销可能变得与进程上下文切换的开销一样大。结果,用线程代替进程的大多数优点都不复存在了。

一种解决方法是采用用户级线程和内核级线程的混合形式,一般称为**轻量级进程**(lightweight processes,LWP)。LWP 运行在单个重量级进程的上下文中,每个进程中可以包含多个 LWP。除了 LWP 以外,系统还提供用户级线程包,向应用程序提供了创建和销毁线程等普通操作。另外,包中还提供了用于线程同步的工具,比如互斥变量和条件变量。重要的是,线程包是完全在用户空间中实现的。也就是说,执行这些线程操作不需要内核的干预。

线程包可以由多个 LWP 共用,如图 3.2 所示。这就意味着每个 LWP 可以运行自己的用户级线程。建立多线程应用程序时首先要创建用户级线程,随后分配每个用户级线程到一个 LWP。将线程分配给 LWP 的行为一般是隐式的,并且向程序员隐藏。

用户级线程和 LWP 一般是通过以下方式结合起来:线程包中有一个用于调度线程的例程。在创建 LWP(一般通过系统调用来创建)的时候,LWP 得到了自己的堆栈,并且执行调度例程,该调度例程会寻找下一个线程来执行。如果有多个 LWP 存在,每一个 LWP 都会执行该调度例程。用来跟踪当前线程集的线程表是由各 LWP 共享的。通过完全在用户空间中实现的互斥标志来对该表进行保护,以确保只能对它执行互斥访问。也就是说,各

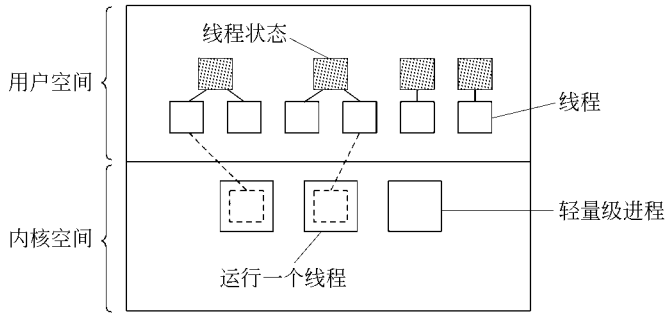

图 3.2 将内核模式的轻量级进程与用户级线程结合使用

LWP 间的同步并不需要任何内核支持。

如果 LWP 找到了一个可运行的线程，它就将上下文切换到该线程。同时其他 LWP 也会寻找其他可运行的线程。如果线程由于互斥变量或者条件变量需要阻塞，它就会在完成必要的管理工作之后调用调度例程。如果找到了另一个可运行的线程，就会将上下文切换成该线程拥有的上下文。这种方法的优点在于，对执行用户级线程的 LWP 来说，用户级线程上下文切换只是普通的程序代码，因为上下文切换完全在用户空间中实现。

现在，我们来看一下线程进行阻塞系统的调用时的情形。在这种情况下，执行过程不再处于用户模式，而是转变到内核模式中，但是仍然在当前的 LWP 上下文中继续执行。在当前的 LWP 无法继续执行的时候，操作系统会将上下文切换到另一个 LWP，这就意味着上下文会重新切换回用户模式。被选中的 LWP 将会简单地从上一次停止的地方继续执行。

将 LWP 与用户级线程包结合起来使用有很多优点。首先，线程创建、销毁及同步化工作的开销相对较小，并且根本不需要内核干预。其次，如果某个进程中有足够数量的 LWP，则阻塞的系统调用将不会导致整个进程被挂起。第三，应用程序并不需要知道 LWP 的存在，它能见到的只是用户级线程。最后，通过在不同 CPU 上执行不同 LWP，LWP 可以在多处理器系统中方便地应用。拥有多个处理器这个事实可以对应用程序完全隐藏。轻量级进程与用户级线程组合的唯一缺点是，仍然必须进行 LWP 的创建和销毁工作，这个工作的开销并不比内核级线程的小。然而值得庆幸的是，只是偶尔需要进行 LWP 的创建和销毁工作，而且受到操作系统的完全控制。

另一种与轻量级进程相似的方法是使用**调度例程激活**（scheduler activation）（Anderson 等 1991）。调度例程激活和 LWP 之间的本质区别在于，在调度例程激活中，当线程在系统调用中被阻塞时，内核对线程包进行**上行调用**（upcall），调用调度例程来选择下一个可执行的线程来执行。当线程释放的时候，也要重复相同的步骤。然而，使用上行调用是一种不太好的做法，因为它破坏了分层系统的结构，在分层系统中只允许对紧接着的下一层进行调用。

3.1.2 分布式系统中的线程

线程的重要特性之一是，它提供了一种方便的方式允许使用会导致阻塞的系统调用而不阻塞该线程所属的整个进程。这种特性使得在分布式系统中使用线程变得特别有吸引

力，因为利用它可以极为方便地将通信表述为同时维护多个逻辑连接的形式。我们将通过对多线程的客户和服务器分别进行详细考察来说明这一点。

3.1.2.1 多线程客户

为了确立高度的分布透明性，在广域网上构建的分布式系统需要隐藏较长的进程间消息传播的时间。在广域网中，传输的延迟很容易达到上百毫秒，甚至几秒。

隐藏通信时间延迟的常规方法是启动通信后立即进行其他工作。这方面典型的例子是Web浏览器。在很多情况下，Web文档是由HTML文件组成的，HTML文件中包含有纯文本文件以及图像组、图标等。为了获得Web文档中每一个组成部分，浏览器必须建立TCP/IP连接，读取输入数据并将数据传递给显示组件。建立连接和读取输入的数据在本质上都是可能导致阻塞的操作。在进行远程通信的时候，还要考虑每个操作所花费的时间可能会相对很长。

Web浏览器一般在开始获取HTML页面后随即就显示它。为了尽量隐藏通信延时，某些浏览器在接收数据的过程中就开始显示这些数据。首先将文本显示出来以便用户查看，并且提供页面滚动之类的功能，同时继续获取组成页面的其他文件，比如图像等。在收到这些文件之后再显示它们。用户不必等待浏览器取得整个页面的所有组件就能够查看页面。

从效果上来说，看起来就好像Web浏览器在同时进行多项任务一样。实践已经证明，以多线程客户的模式来开发浏览器可以显著地使问题得到简化。只要取得了主HTML文件，就可以激活多个独立的线程，它们分别负责取得页面的各个部分。每个线程都与服务器建立一个独立连接以获取数据。只要假定进行导致阻塞的调用不会将整个进程挂起，与服务器建立连接和读取数据的过程就可以使用标准的（可能导致阻塞的）系统调用来编制。(Stevens 1998)中也说明了，用于每一个线程的代码都是相同的，并且首先是简单的。在浏览页面期间，用户只会注意到图像等内容显示上的延迟，但是可以对文档进行浏览。

在可以同时打开多个连接的情况下，使用多线程的Web浏览器还有另一个明显的好处。在上面的例子中，建立了到同一个服务器的多个连接。但是，如果该服务器的负载过重，或者比较慢，这种方法与逐个获取组成页面的文件的方法相比，在性能上并不会有什么真正的提升。

然而，在许多情况下，Web服务器会复制到多台机器上，每个服务器负责提供一组完全相同的Web文档。复制的服务器位于同一个站点，并且名字是相同的。当对Web页的访问请求到来时，该请求被转发到其中的一个服务器，转发到哪一个服务器的决策通常是通过循环策略或者某些其他负载平衡技术做出的(Katz 等 1994)。在使用多线程客户的时候，可以与不同的服务器副本建立连接，这样就可以并行地进行数据传输了，并且确保整个Web文档完全显示出来所需的时间与使用无复制的服务器的情况相比要短得多。这种方案只在客户真正能够处理输入数据的并行流时才能够发挥作用，而使用线程正是达到这个目的的理想方式。

3.1.2.2 多线程服务器

虽然我们已经看到多线程客户有许多优点，但是分布式系统中多线程技术主要还是在

服务器端发挥作用。实践表明，多线程技术不仅能够显著简化服务器代码，还能够使得应用并行技术来开发高性能的服务器变得更加容易，即使在单处理器系统上也是如此。而且，目前多处理器计算机广泛地用作通用工作站，所以多线程技术就更加有用了。

为了理解使用线程给编写服务器代码带来的好处，考虑一下文件服务器的组织结构，该文件服务器可能会偶尔由于等待磁盘操作而阻塞。文件服务器一般等待输入的文件操作请求，随后执行该请求，最后送回应答。其中一种特别流行的组织结构如图3.3所示。图中有一个称为**分发器**(dispatcher)的线程，由它来读取文件操作请求。客户发送请求到服务器的某个已知端点。在对请求进行检查以后，服务器选择一个空闲的(也就是阻塞的)**工作者线程**(worker thread)，由它来处理该请求。

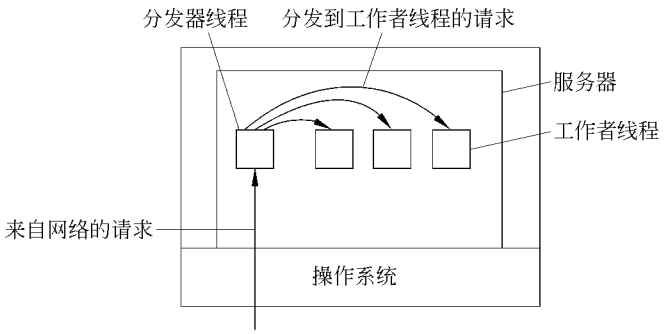

图3.3 以分发器/工作者模型组织起来的多线程服务器

工作者线程在本地文件系统上执行阻塞的read调用，执行该调用将会导致该线程被挂起直到数据从磁盘上读出为止。如果该线程被挂起了，就选择另一个线程接着执行。比如说，可以选择执行分发器线程以完成更多工作。此外，也可能选择另一个准备就绪的工作者线程来运行。

现在来考虑在不使用多线程的情况下文件服务器的编写方式。一种可能是让服务器作为单个线程来工作。由文件服务器主循环程序来获得请求，检验该请求，随后执行该请求直至完成，然后再接受另一个请求。在等待磁盘操作的时候，服务器是空闲的，但也不对其他请求进行处理。也就是说，来自其他客户的请求得不到处理。另外，如果文件服务器运行于专用机器上(大多数情况下是如此)，文件服务器等待磁盘操作的时候CPU将会完全空闲。这样做的结果是，服务器每秒能够处理的请求数目大大减少。而使用多线程能够显著提升性能，而且每个线程都是以通常的方式编写成顺序执行的。

至此我们已经看到了两种可行的设计方法：多线程文件服务器以及单线程文件服务器。假定无法使用多线程，但是系统设计者又发现使用单线程所带来的性能损失是不可接受的，那么第三种可行方法是将服务器作为一个大的有限状态机来运行。当请求输入的时候，由唯一的线程对请求进行检查，如果可以利用缓存中的内容来满足请求当然好，如果缓存的内容无法满足就必须向磁盘发出消息，请求执行磁盘操作。

然而，向磁盘发出消息之后线程并不阻塞，而是在状态表中记录下该请求的当前状态，随后接收下一个消息。下一个消息可能是一个开始新工作的请求，也可能会是磁盘对之前的操作请求所做的应答。如果是新工作请求，就进行该工作。如果是来自磁盘的应答，就从表中取出之前记录下的那个请求的有关信息，对应答进行处理并把结果发送给客户。如果

采用这种方法，服务器必须使用非阻塞性的 send 和 receive 调用。

在这种设计中，前两种方法使用的“顺序处理”模型不见了。对于每一个发送和接收的消息来说，计算状态都必须显式地保存下来，存储在表中。从效果的角度来看，这么做是在以一种复杂的方法模拟线程及其堆栈。进程是作为有限状态机来运行的，它在接收到一个事件之后根据其内容来决定采取的操作。

现在，多线程的作用已经很清楚了。多线程能够保留顺序处理的思路，使用阻塞性的系统调用（比如，RPC 与磁盘通话），并且仍然能达到并行处理的目的。使用阻塞性系统调用能使编程更加容易，而并行处理能够提升性能。单线程服务器保留了使用阻塞性系统调用所带来的方便之处，但是放弃了性能。而有限状态机的方法能够通过并行获得高性能，但是由于使用非阻塞性系统调用，编程上比较困难。这些模型总结在图 3.4 中。

模型	特征
多线程	并行，使用会导致阻塞的系统调用
单线程进程	非并行，使用会导致阻塞的系统调用
有限状态机	并行，使用非阻塞系统调用

图 3.4 构建服务器的 3 种方式

3.2 虚 拟 化

线程和进程提供了一种同时做多件事情的方式。从效果上来说，它们可以用来构建看起来像同时执行的程序。在单处理器的计算机上，这种同时执行当然只是一种感觉。由于只有一个处理器，任一时间只能是某个线程或进程一条指令在执行。通过在线程和进程之间快速切换，给人的感觉是多个线程和进程在同时执行。

这种只有单个处理器但感觉有多个处理器机制可以扩展到其他资源，导致所谓的**资源虚拟化**（resource virtualization）。这种概念在过去几十年里也应用过。但只是当分布式计算机系统更广泛的使用并且更复杂后，虚拟化才重新变得热起来。这也表明应用软件通常比底层的系统软件和硬件更有生命力。这一节中我们考察虚拟化的作用并且讨论它的实现。

3.2.1 虚拟化在分布式系统中的作用

如图 3.5(a)所示，基本上每个分布式计算机系统都提供一个给上层软件的界面。界面可以有多种，从处理器提供的基本指令集到很多中间件系统提供的巨大的应用程序界面集。如图 3.5(a)所示，虚拟化本质上是扩展或替换一个现存界面来模仿另一个系统的行为。我们将会讨论虚拟化的技术细节。但首先考察一下为什么虚拟化对分布式系统很重要。

在 20 世纪 70 年代提出虚拟化的最重要原因是让老的软件可以在新的昂贵的大型机上运行。那些老的软件不仅包括应用软件，也包括了运行这些软件的操作系统。这种支持老软件方法在 IBM370 和后继机上得到成功的应用。IBM370 和后继机提供一个虚拟机并把多个不同的操作系统移植到它上面。

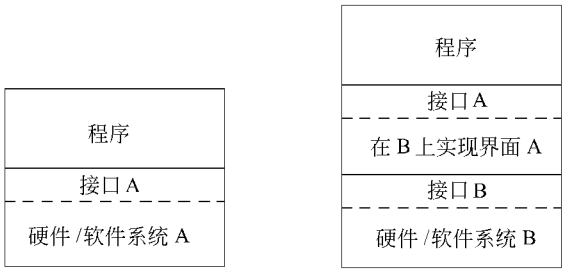

图 3.5

(a) 程序、接口与系统之间的组织结构；(b) 系统A位于系统B之上的组织结构

随着硬件变得便宜，计算机功能更强大，并且操作系统种类变少，虚拟化也变得不那么重要。然而，从20世纪90年代期以来，有几个原因促使事情又一次发生了变化。

第一，尽管硬件和低层系统软件变化得比较快，高层软件（比如中间层和应用软件）却稳定得多。也就是说，我们面对的一种情况是旧有软件（legacy software）的维护跟不上下层平台更新的步伐。通过移植旧有软件的底层接口到新平台，虚拟化可以帮助解决这个问题。另外，这也使得一大类的现有软件可以在立刻新平台上用。

另一个同等重要的事实是网络已经无所不在。很难想象有一台没有联网的现代计算机。在实践中，这种无所不在的网络互联使得系统管理员要维护很多不同的服务器。每台服务器运行不同的应用程序，可以被客户访问。同时多种资源应该很容易地被应用程序访问。虚拟化可以提供很大帮助：通过让每个应用程序运行在自己的虚拟机上（可能包括运行在一个通用平台上的相关的库和操作系统），平台和机器的种类可以减少。

后一种虚拟化提供了高度的移植性和灵活性。例如，为实现能轻易支持动态内容（content）复制的内容传输（content delivery）网络，（Awadallah 和 Rosenblum 2002）论证，如果边界服务器（edge server）能支持虚拟化，允许动态复制一个站点（site）及其环境，管理将会容易得多。如我们将要讨论的一样，主要是这些移植性的好处使得虚拟化成为分布式系统的重要机制。

3.2.2 虚拟机体系结构

在实践中有多种实现虚拟化的方法。（Smith 和 Nair 2005）简要描述了这些方法。为理解不同的虚拟化技术，首先要认识到计算机系统通常在4个不同层次提供4个不同界面：

(1) 由**机器指令**（machine instruction）组成，可由任何程序激起的硬件软件界面。

(2) 由机器指令组成，只有特权程序（像操作系统）才可激活的硬件软件界面。

(3) 由操作系统提供的**系统调用**（system call）组成的界面。

(4) 由库调用组成的界面，通常形成了所谓的**应用程序编程接口**（application programming interface，API）。很多情况下前述的系统调用由API隐藏。

图3.6显示了这些不同的界面。虚拟化的实质是模仿这些界面的行为。

虚拟化可采用两种方式。第一，可以构建一个运行时（runtime）系统，实质上提供一套抽象指令集来执行程序。指令可以被翻译执行（像Java运行环境），也可以仿真执行，就像

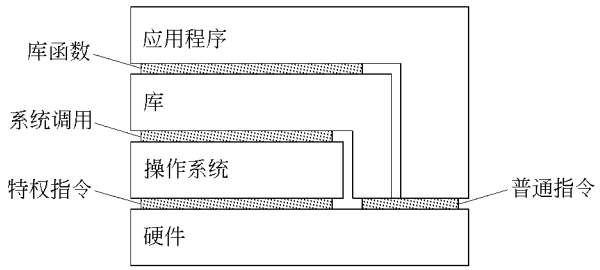

图 3.6 计算机系统多种界面

在 Unix 平台上运行 Windows 应用程序。在后一种情况，仿真器必须模仿系统调用的行为，而这已被证明远非寻常。这种虚拟化被 Smith 和 Nair 称为**进程虚拟机**(process virtual machine)，用以强调虚拟化实质上作用在单个进程上。

另一种虚拟化方式是提供一种系统。把它做成一层完全屏蔽硬件但提供一个同样指令集(或其他硬件)的界面。关键是这个界面可以同时提供给不同的程序。结果，可以有多个不同的操作系统独立并发地运行在同一平台。这个层通常叫**虚拟机监视器**(virtual machine monitor，VMM)。这种方法的典型例子是 VMware (Sugerman 等，2001)和 Xen (Barham 等，2003)。这两种不同方法显示在图 3.7 中。

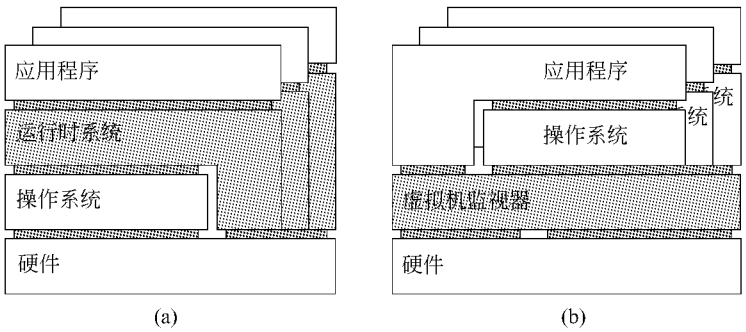

图 3.7

(a) 进程虚拟机，混合多个应用程序和运行时系统；(b) 虚拟机监视器，混合多个应用程序和操作系统

像 Rosenblum 和 Garfinkel 争论那样，虚拟机监视器将会变得对分布式系统的可靠性和安全越来越重要。由于它们能分离一个应用程序和它的环境，由错误或安全攻击引起的失败不再会影响整台机器。另外，像我们以前提过，由于虚拟机监视器进一步分离了硬件和软件，允许一个完整环境从一台机器移植到另一机器，移植性得到很大改善。

3.3 客 户

在前面的章节中我们讨论过客户-服务器模型、客户和服务器所扮演的角色以及它们之间的交互。让我们对客户和服务器分别进行详细剖析，在本节中我们先讨论客户，服务器将在下一节中讨论。