# 进程

在这一章中，我们将研究各种不同类型的进程是如何影响分布式系统工作的。进程的概念起源于操作系统中，它是用来定义一个程序的运行的。在操作系统层面，进程的管理和调度可能是最重要的课题。然而，当在分布式系统中讨论时，进程的一些其他课题变得更为重要。

我们首先重点讨论线程和它在分布式系统的作用。事实证明，线程不但在多核多处理器环境中对性能有重要的作用，更有助于构造客户端和服务端。在很多情况下，我们看到线程被进程所替代，并且使用底层操作系统来保证保护和促进通信。然而，当性能受到威胁时，线程将扮演着重要的角色。

近些年，虚拟化的概念开始流行起来。虚拟化允许应用程序及其完整的环境(包括操作系统)与其他应用程序并发运行，但高度独立于底层硬件和平台，从而实现高度的可移植性。另外，虚拟化可以帮助隔离由故障和安全引起的错误。这对于分布式系统是很重要的概念，我们将单独讨论。

客户-服务器组织架构在分布式系统中很重要。在这一章节，我们将仔细研究客户端和服务端的典型组织。我们将注意到服务端的典型设计，包括那些在object-based分布式系统中使用的类型。一个应用很广泛的Web服务是Apache，我们将单独了解。服务端的集群组织也很重要，特别是他们需要合作得像是单一系统在工作一样。我们将介绍类似的例子，包括广域服务PlanetLab。

在广域分布式系统中，一个重要的课题是机器间的晋城移动。进程转移或者代码转移可以帮助实现可伸缩性，或者帮助动态配置客户端和服务端。关于代码转移还有其如何实现的，我们将在这一章节讨论。

## 3.1线程

虽然进程组成了分布式系统的基本单元，但实践表明，分布式系统以进程作为最小粒度并不足够充分（就是还有线程呗）。作为替代，有进程控制的多个线程组成了更小的粒度，这能更好的构建分布式应用并获得更好的性能。在这一节中，我们将关注在分布式系统中的线程是如何发挥作用的并解释它为什么这么重要。关于线程以及如何通过线程构建应用可以参考

[Lewis and Berg, 1998; Stevens, 1999; Robbins and Robbins, 2003]。这些很好的从概念上诠释了多线程并发程序。

**线程的介绍**

为了理解分布式系统中的线程，我们需要首先理解进程是什么，并且进程和线程是什么关系。为了运行一个程序，操作系统会创造一系列虚拟进程，每一个会运行不同的程序。为了跟踪这些进程，操作系统有一个进程对照表，包括存储CPU的寄存器只，内存映射，打开文件，计数器，privileges，等。这些条目构成了进程的上下文。

进程上下文可以看作是硬件处理器上下文的软件模拟。硬件处理器上下文包括硬件处理终端，以及存储CPU处理现场的最小信息。这个进程上下文包括程序计数器，或者其他的寄存器值比如栈指针。

进程通常定义为一个运行的程序，也即正在操作系统虚拟处理器上运行的程序。一个关键问题是，操作系统会很小心的保证独立的进程之间不会相互干扰的彼此。换句话说，多个进程共享同一个cpu以及其他硬件资源这件事，是透明的。通常，操作系统需要硬件的支持来完成这种分割操作。

这种透明性是有代价的。比如，当一个进程被创建时，操作系统必须创建一个完全独立的地址空间。这种分配可以意味着初始化内存段，例如，将数据段归零，将相关程序复制到文本段，并为临时数据建立堆栈。这样，在两个进程间切换CPU需要消耗性能。除了在寄存器临时存放数据外（包括程序寄存器和栈指针），操作系统同样会修改内存管理单元（MMU）的寄存器和使地址转换缓存无效，例如在转换后备缓冲区(TLB)中。另外，如果操作系统需要支持超过他内存大小的进程数量，那么需要在内存和磁盘间进行内存交换。

像进程一样，线程也会执行一段代码，并独立于其他线程。然而，不同于进程，如果实现并发透明性会导致性能下降，则不会尝试实现高度的并发透明性。因此，线程系统通常保持最小的信息来允许CPU可以被其他线程共享。特别是，一个**线程上下文**通常和进程上下文的组成没有区别，除了一些关于线程管理的信息。例如，线程系统可以跟踪一个线程当前在互斥量变量上被阻塞的事实，以便不选择它执行。关于其他线程的不必要的信息是不必理会的。因此，保护数据不受单个进程内线程的不适当访问完全留给应用程序开发人员。因为我们发现一个处理器的上下文包含在线程上下文中，而线程上下文包含在进程上下文中。

正如我们刚才概述的，部署线程有两个重要含义。首先，多线程应用的系统性能很难比同样的单线程系统性能差。实际上，多线程可以获得更多的性能。其次，由于线程不能自动的像进程一样保护自己不受其他线程干扰，多线程应用的独立需要更多的智慧和努力。合适的设计和事情的简单化是有帮助的选择。不幸的是，从目前的实践来看，这个原则并不好理解。

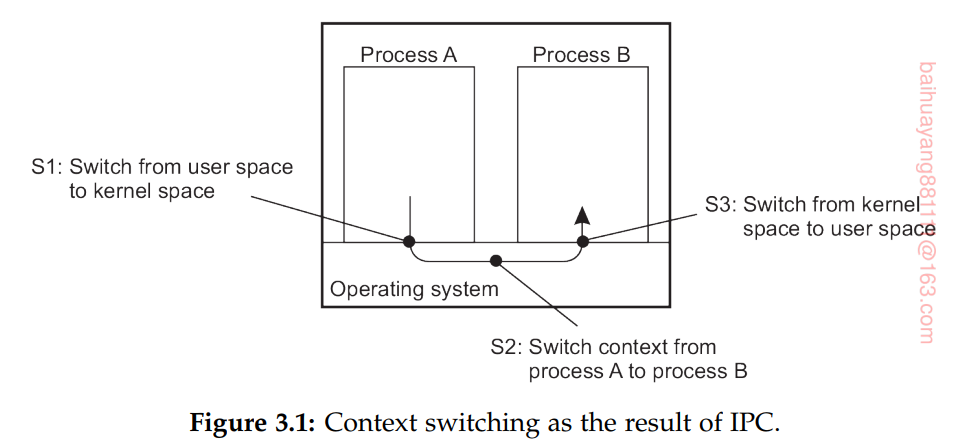
**在非分布式系统中的线程使用**

在讨论分布式系统中线程的功能之前，我们首先讨论在非分布式系统中线程的功能。使用多线程系统有很多好处。

最重要的一个优势是，作为但线程进程，当一个阻塞方法被调用时，进程将会被阻塞。为了说明这一点，考虑一个应用程序，例如电子表格程序，并假设用户持续地、交互地想要更改值（举个访问数据库的栗子更好，比如需要向数据库刷10分钟的数据，进程将阻塞10分钟）。电子表格程序的一个重要特性是，它维护不同单元格之间的功能依赖关系，这些单元格通常来自不同的电子表格。因此当一个单元更改时，其他依赖的单元会自动更新。当用户更改一个单元时，这种更改会出发一系列计算。如果只有一个进程进行控制，在程序等待输入之前，计算是不能够完成的。像这样，当依赖的计算完成之前，提供数据是不容易的。一个简单的处理方式是，获取两个线程：一个负责处理用户输入，另一个负责更新。同时，第三个线程将数据备份。

另一个多线程的优点是当在一个多处理器或者多核系统上运行程序是，可以进行并行计算。当数据都被存储在共享的内存时，每一个线程都被分配给了不同的CPU和处理器。如果涉及得当，这种并行计算可以是透明的：这个进程同样可以在单核系统中运行，尽管这样运行会比较慢。在多核和多处理器价格变得相对便宜的情况下，多线程的并行处理变得很重要了。这种计算机系统通常发生在客户-服务端系统的服务端，但目前也广泛应用于智能手机等设备。

多线程同样在大型应用的上下文中很有作用。这些应用通常由一些相互合作的程序，即独立执行的进程组成。这个方式是典型的Unix 环境的方式。程序之间的合作通过**interprocess communication（IPC）**来实现。对于Unix系统，这些机制包括，管道，消息队列和共享内存等。这种IPC机制的主要缺点是上下文切换很消耗资源，图3.1介绍了这些机制。



除了使用进程，一个应用还可以通过不同的独立线程来构成。他们之间的通信完全通过共享数据来完成。线程切换有时可以只在用户态完成，在一些其他的实现中，内核也会参与线程并进行调度。这样会极大的提高性能。最终，这里有一个使用线程的纯粹的软件工程的原因：许多由线程构成的应用会变得更简单。考虑需要执行多个(或多或少独立的)任务的应用程序，如前面讨论的电子表格示例。

**线程实现**

线程通常以线程包的形式提供。这种包包括线程的创建和摧毁操作，同样包括线程同步相关的锁和条件变量等。这里主要有两种方法来实现线程包。第一种方法是构成完全在用户控件执行的一个线程库。第二种方法是，使用内核感知这些线程并调度线程。

用户级别的线程库有一系列好处。第一，创建和摧毁线程都不是很好资源。因为所有的线程权限都在用户地址区域，创建线程唯一值得考虑的就是在栈空间分配内存。同样的，摧毁线程主要包括在栈空间释放内存，但现在已经不再使用这种方式了。这两种方式都很廉价。

第二个使用用户级别线程的优点是上下文切换可以在几个指令下完成。基本上，只有CPU寄存器的值需要保存并且接下来当线程切换时重新加载之前存储的值。这里不需要改变内存映射，冲刷TLB，或做CPU计数等。然而，如Note3.1讨论的，上下文切换的大部分开销是由扰乱内存缓存造成的。

用户级线程的一个主要缺点是使用了**多对一的线程模型**：即多个线程映射为一个调度实体。作为一个结果，调用阻塞系统调用将立即阻塞线程所属的整个进程，从而阻塞该进程中的所有其他线程。正如我们解释的，线程通常对于构建由许多在逻辑上同时执行的小组成部分构成的大型应用是很有帮助的。在这种情况下，在同一时刻，IO阻塞不应该阻止其他部分的执行。对于这种应用，用户级别的线程是不会起到作用的。（注意这里是用户级别的，我们平常用的是内核级别的）

这个问题可以通过内核级别的实现来解决，也即使用**一对一线程模型**，即一个线程为一个调度实体。当然，每个线程的操作都是由操作系统内核负责，每一次都需要系统调用。这里线程上下文切换看起来会比进程的上下文切换更耗资源。然而，考虑到上下文切换的性能通常是由内存缓存的无效使用决定的，而不是由多对一或一对一线程模型之间的区别决定的，许多操作系统现在提供了后者模型，即使只是为了简单。

最后一点需要注意的是，必须认识到使用线程是组织应用程序中同时执行和并发执行的一种方式。在实践中，我们经常看到应用程序被构造为一组并发进程，共同使用操作系统提供的进程的设施。一个很好的例子就是ApacheWeb服务器，它是通过一系列进程来处理请求的。每个进程都组成一个单一的线程实例，但是都能通过标注你的方法与其他实例通信。

正如Srinivasan讨论的，使用进程代替线程对于将数据空间分离有很大的好处：每个线程都在独立的自己的数据区工作，并在操作系统层面与其他的进程隔离。这种分离的好处不应低估：线程编程被认为是出了名的困难，因为开发人员完全负责管理对共享数据的并发访问。使用进程，数据空间最终受到硬件支持的保护。如果一个进程访问了它自己之外的内存，硬件将会抛出一个异常，这个异常将在之后被操作系统处理。在线程则没有这种待遇。

**分布式系统的线程**

线程的一个重要性质是，它能够提供方便的手段使得在系统中的阻塞调用，不影响该线程所属进程的运行。这个性质使得线程在分布式系统中的使用变得很有吸引力，因为它使得同时维持多个逻辑通信的连接变得容易。为了说明这个观点，我们将分别观察多线程的客户-服务端系统。

**多线程客户端**

为了建立高度透明的分布式系统，在广域网中运行的分布式系统可能需要隐藏较长的进程间消息传播时间。广域网中的往返延迟很容易达到几百毫秒，有时甚至是几秒钟。

隐藏通信延迟的通常方法是启动通信并立即处理其他事情。这在Web浏览器中很常见。在很多情况下，一个Web文档包含很多个HTML文件如图片，标签等。为了获取这些元素，浏览器需要建立TCP/IP连接，读取传入的数据，并将它发送到需要展示的区域。建立连接或者读入数据通常都是阻塞操作。在处理长途通信时，我们也有一个缺点，每个操作完成的时间可能相对较长。

一个Web浏览器通常都是抓取HTML页面然后展示它。为了尽量隐藏通信延迟，一些浏览器开始会一边接受数据，一边进行展示。当文本对用户可用，或者使用滚动工具等，浏览器将继续获取组成页面的其他文件，图像等。这些文件和图像将被展示。用户不用等到页面所有组件都被获取才能看到页面。

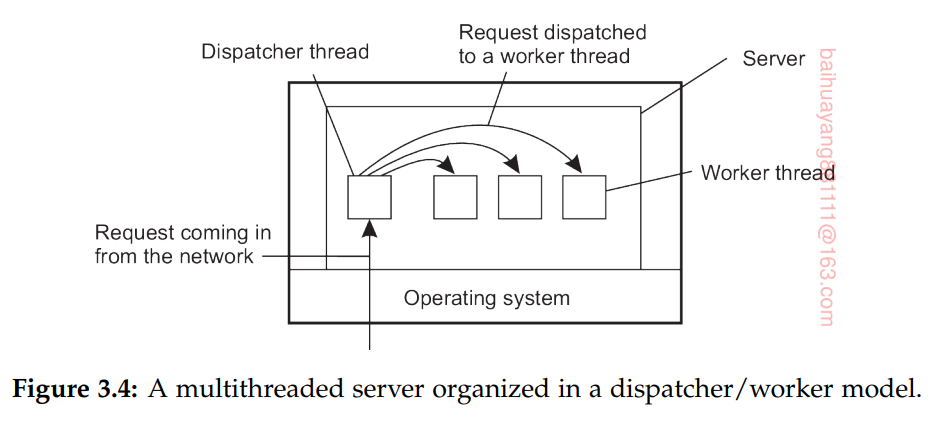
实际上，Web浏览器在同时做一系列任务。事实证明，将浏览器开发为多线程客户机大大简化了工作。当主要的HTML文件被获取时，其他的线程会被激活并获取其他部分。每个线程都单独和服务器建立连接并抓取数据。建立连接和读取数据可以使用标准系统调用，这里假定一个阻塞调用不会暂停所有的进程。同样Stevens提到，每个线程的代码都很简单。与此同时，用户只注意到图像显示的延迟等等，但是可以浏览文档。Web浏览器使用多线程的另一个好处是可以同时打开多个连接。在之前的例子中，和服务端建立了多个连接。如果这个服务器超负荷了，那么这种获取方式将不会提高性能。

然而，在许多例子中，Web服务器会复制到多个机器，每个服务器提供相同的Web文档。这些复制的服务器位于同一个站点，并且都叫同一个名字。当一个Web请求到来时，这个请求会来到其中的某一个服务器，这通常会使用轮询或者其他负载均衡的技术。当使用多线程客户端时，连接会在多个副本上建立，允许数据并行传输，这与不适用复制服务器相比，整个Web文档的传输要快很多。只有当客户机能够处理真正并行的传入数据流时，这种方法才有可能。线程是实现此目的的理想方法。

**多线程服务器**

虽然多线程在客户端具有很好的优势，但更主要的应用是放在了服务端。实践表明，多线程不仅大大简化了服务器代码，而且使开发利用并行性获得高性能的服务器变得更加容易，甚至在单处理器系统上也是如此。然而，随着现代多核处理器的发展，多线程并行成为了一种主要的实现方式。

为了了解服务端多线程的有点，考虑一个偶尔会因为硬盘等待儿阻塞的文件系统。这个文件服务器通常会等待一个输入请求，进而获取请求，然后返回一个应答。如图3.4所示。这里一个线程，或者说是**分配器**，读取进入的请求。紧接着请求被发送到服务器。检查这个请求后，服务器挑选一个空闲的**工作线程**来处理请求。



这个工作线程会执行一个本地读取操作，这是阻塞的，可能会使得获取数据的线程暂停。如果线程暂停，其他的线程将会继续执行。这样，分配线程可以执行更多的工作。作为替代，另一个工作线程会被选择来继续运行。

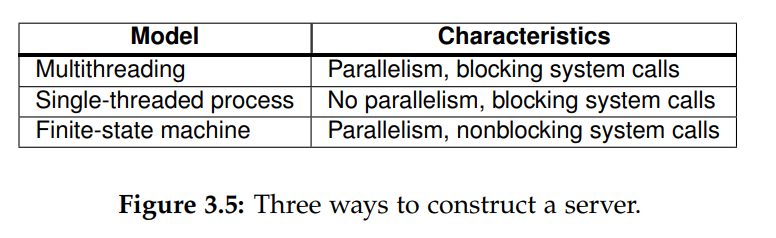
现在考虑服务没有使用线程的情况。一个可能就是讲服务器处理为单线程了。主要的循环即为获取请求，检测，处理返回，然后再进行下一个请求。当在硬盘上等待时，服务器是空闲的并且没有处理其他的请求。通常，其他客户端的请求不会被处理。此外，如果文件服务器运行在专用机器上(通常是这种情况)，那么在文件服务器等待磁盘时，CPU只是处于空闲状态。这样处理的结果是更少的请求会被执行。线程虽然获得了可观的性能，但是每个线程都是顺序执行的。

到目前为止，我们看到两种设计：多线程文件服务和单线程文件服务。第三种选择是将服务器作为大型单线程有限状态机运行。当请求到达时，唯一的线程将会检测它。如果可以从内存缓存中得到满足，那很好，但是如果不能，线程必须访问磁盘。然而，和磁盘阻塞不同的是，线程会调度一种异步磁盘操作，即之后会被操作系统中断。为了使它工作，线程会记录请求的状态，并持续观察之后是否会有其他的请求。（不是多个线程，而是单一线程接收请求，系统调用一个神奇的异步操作访问硬盘，之后发出中断来通知线程。）

当一个磁盘操作完成时，一个操作系统将会通知线程，并查找关联的请求状态并进一步执行它。最终，一个响应将会被一个非阻塞网络调用返回给客户端。

在这个设计中，我们在前两种情况下拥有的“顺序过程”模型丢失了。每次线程需要执行阻塞操作时，它都需要准确地记录它在处理请求时的位置，还可能存储其他状态。一旦完成这些工作，就可以开始操作并继续进行其他工作。其他工作意味着处理新到达的请求，或者处理先前启动的操作已经完成的请求。当然，如果没有其他工作，线程将会阻塞。实际上，我们正在艰难地模拟多线程及其各自堆栈的行为。该流程作为一个有限状态机进行操作，该有限状态机获取一个事件，然后根据其中的内容对其做出反应。

现在我们能明白线程提供了什么好处了。它可以同时实现如同进程一样的顺序执行或者执行并行。阻塞系统调用使编程变得更容易，因为它们看起来就像普通的过程调用一样。另外，多线程允许并发执行并提高性能。单线程服务器保留了阻塞系统调用的易用性和简洁性，但就每个时间单元可以处理的请求数量而言，它可能严重阻碍性能。有限状态机方法通过并行实现高性能，但使用非阻塞调用，这通常很难编程，因此很难维护。这些模型的总结如图3.5.



同样，我们也可以使用多进程来代替多线程。它的优势是操作系统可以提供进程维度的数据隔离。而，如果进程需要进行大量通信，那么与使用线程相比，我们可能会看到对性能的明显负面影响。

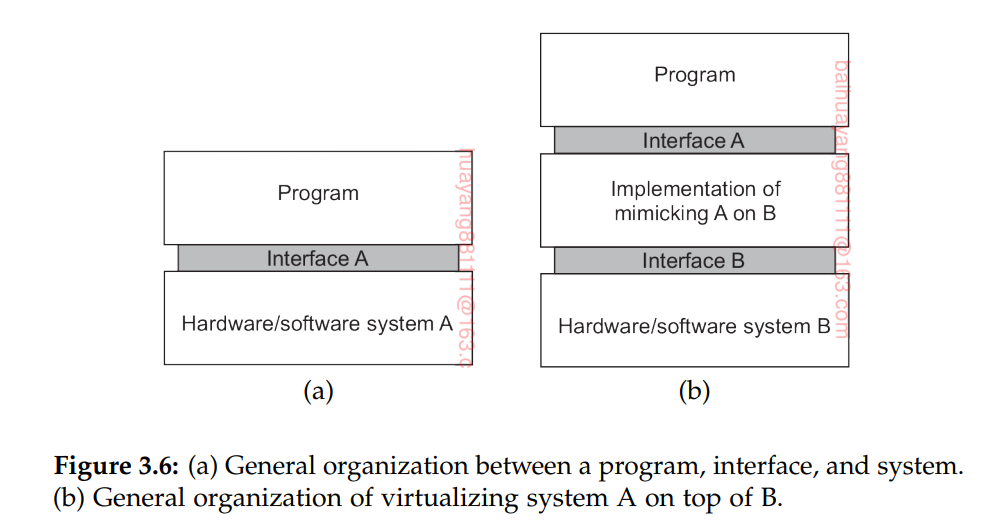
## 3.2虚拟化

线程和进程可以在同一时间做很多事情。实际上，他们允许我们构建同时执行的程序。在一个单进程的机器上，这种同时执行其实是一种错觉。因为只有一个单核cpu，所有只有一个单线程或者进程的设施会执行。通过快速的切换进程和线程，这种并发的错觉将创建。

这种使单个cpu分开运行，并加装可以同时运行多个资源的技术，称之为**资源虚拟化**。虚拟化已经被应用了许多年，但由于分布式计算机系统变得更加大众和复杂，使得人们对虚拟化又重新燃起了兴趣，这也使得应用软件可以在这种软件和硬件系统上存在更长的时间。

**虚拟化原则**

原则上，每个分布式系统都会提供一个给高级软件的编程接口，如图3.6(a)。这里有很多不同的接口类型，从CPU提供的基本指令集到许多当前中间件系统附带的大量应用程序编程接口集合。从本质上讲，虚拟化处理扩展或替换现有接口，以便模拟另一个系统的行为，如图3.6(b)。我们将简短的讨论虚拟化的技术细节，但让我们首先讨论虚拟化的重要性。



**虚拟化和分布式系统**

70年代有一个很重要的介绍虚拟化背景的原因，即是否允许遗留软件在昂贵的主机硬件上运行。这样的软件不仅仅包括各种各样应用，但实际上也包括他们依赖的操作系统。这种支持遗留软件的方法已经成功地应用于IBM 370大型机(及其继任者)上，这些大型机提供了一个虚拟机，将不同的操作系统移植到其中。

当硬件变得越来越便宜，计算机变得越来越强大，并且操作系统的种类越来越少，虚拟化的话题则越来越少。然而，在90年代，这个问题再次被提及。首先，虽然硬件和底层系统软件变化相当快，但是更高抽象级别的软件(例如，中间件和应用程序)通常要稳定得多。换句话说，我们面临的情况是，遗留软件不能与它所依赖的平台保持相同的步调。虚拟化可以将遗留接口移植到新平台，从而立即为现有程序的大型类开放新平台，从而在这方面提供帮助。

同样重要的是，现在网络已经是无处不在的了。很难想象一个现代的计算机没有连上网络。实际上，这种连接要求系统管理员维护大量异构的服务器计算机集合，每台服务器计算机运行非常不同的应用程序，客户机可以访问这些应用程序。同时有很多的资源可以同时被这些应用访问到。虚拟化可以帮助这一点：可以通过让每个应用程序在自己的虚拟机上运行(可能包括相关的库和操作系统，而这些库和操作系统又运行在一个公共平台上)来减少平台和机器的多样性。

最后一种虚拟化提供了高度的可移植性和灵活性。例如，为了实现能够轻松支持动态内容复制的内容交付网络，Awadallah和Rosenblum[2002]认为，如果边缘服务器支持虚拟化，允许动态复制整个站点(包括其环境)，管理就会变得容易得多。这些论点仍然有效，实际上，可移植性可能是虚拟化在许多分布式系统中扮演如此关键角色的最重要原因。

**虚拟化类型**

虚拟化可以通过很多方式来实现。Smith和Nair对这些不同的方法进行了概述。为了理解虚拟化的不同，重要的是理解计算机系统大体上为我们三个层级的四种不同类型的接口：

1. 硬件和软件之间的接口，称为指令集体系结构(ISA)，形成一组机器指令。这个集合被分成两个子集：

•特权指令，仅允许由操作系统执行。

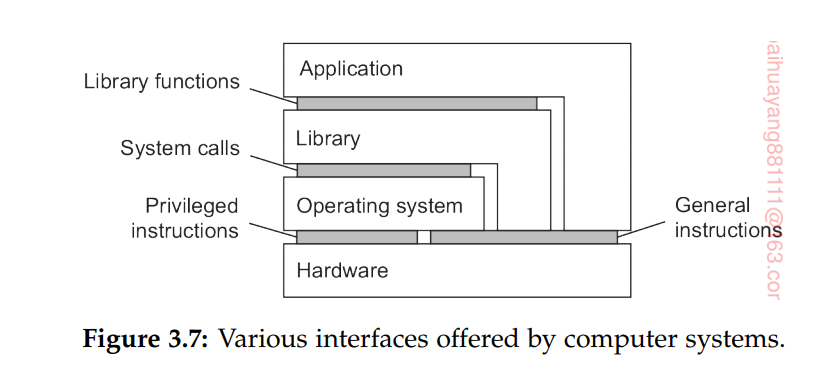
•通用指令，可由任何程序执行

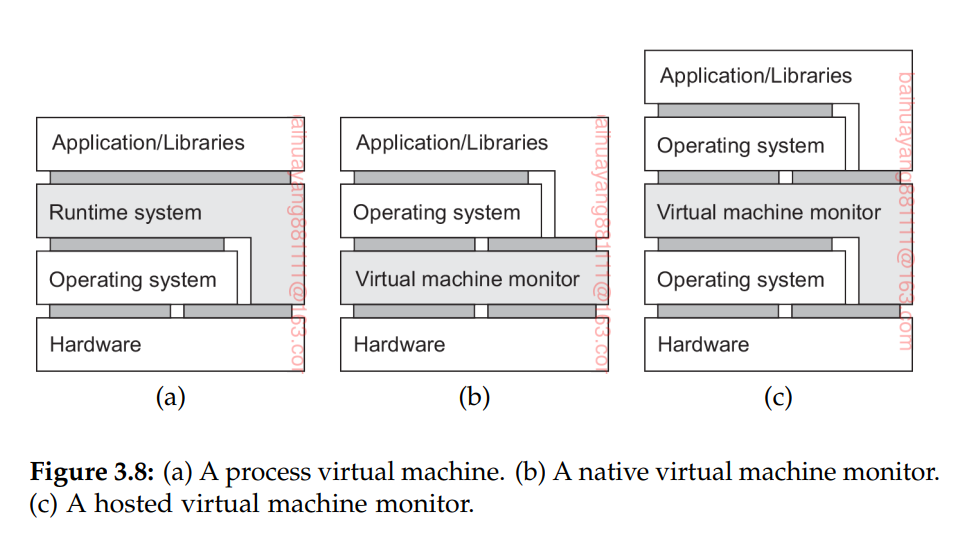
2. 由操作系统提供的系统调用组成的接口

3. 由库调组成的接口，并由此组成的**应用程序接口(API).**在许多情况下，前面提到的系统调用是由API隐藏的。

这些不同的类型如图3.7所示。虚拟化的本质是模拟这些接口的行为。

虚拟化可以发生在两种不同的情况。第一，我们可以构建一个运行时系统，这个系统本质上提供了一组抽象的指令集为执行应用调用。指令可以被解释(Java运行时环境就是这样)，但是也可以像在Unix平台上运行Windows应用程序那样进行模拟。请注意，在后一种情况下，仿真器还必须模拟系统调用的行为，这一点已被证明是非常重要的。这种虚拟化类型如图3.8(a)所示，Smith和Nair称之为**process virtual machine**，强调了虚拟化只是针对一个进程。



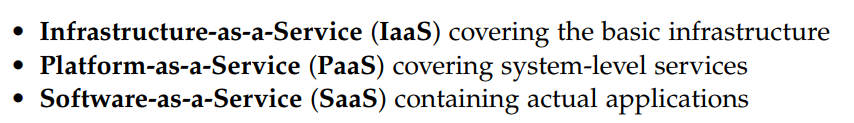


一个替代的方法如图3.8b所示，在原始硬件上提供一层，并提供相同的指令集。这就是**native virtual machine monitor**。它被叫做本地的是因为它直接在硬件上层实现。注意，虚拟机监视器提供的接口可以同时提供给不同的程序。因此，现在可以在同一个平台上独立并发地运行多个不同的客户机操作系统。本地虚拟机监视器会提供不同资源的正常访问，比如外部存储和网络。像任何一个操作系统一样，它需要实现这些资源的设备驱动。除了上面的方法，还有一种 建立在受信任的**主机操作系统**上的**主机虚拟化机器监视器**，如图3.8(c)所示。在这种情况下，虚拟机监视器可以使用该主机操作系统提供的现有设施。它通常必须被赋予特权，而不是作为用户级应用程序运行。使用托管虚拟机监视器在现代分布式系统(如数据中心和云)中非常流行。

正如Rosenblum和Garfinkel讨论的，虚拟机器对于分布式系统的稳定性和安全性都十分重要。因为他允许应用和环境保持独立，一个错误的发生或者一个安全攻击都不会对一整个计算机产生影响。另外，正如我们之前提到的，由于虚拟机提供了一个硬件和软件的解耦，可移植性被大大提高了。我们将在3.5节讨论。

**分布式系统虚拟机的应用**

从分布式系统的特性来看，虚拟化应用的最重要的部分就是云计算。正如我们再1.3节提到的，云计算提供了三种不同类型的服务



虚拟化在IaaS中很重要。不需要出租一个物理机，云计算只需要出租物理机的一部分-即虚拟机给用户。这种虚拟化的优雅就在于使得各个用户之间保持独立性，而用户却好像有独占一个物理机的错觉。独立性不是完整性。如果实际的物力资源被共享，就会导致性能的下降。

为了使问题更具体化，我们考虑亚马逊的**Elastic Compute Cloud**或者**EC2**。EC2允许创建在环境上连续的几个联网的虚拟机，这样组成一个分布式系统。为了简化工作，有大量预配置的机器映像可用，称为Amazon机器映像，或者简称为**AMI**。AMI是一个可安装的软件包，由一个操作系统内核和一些服务组成。一个简单的例子就是**LAMP**，包括Linux 内核，Apache Web服务，Mysql数据库系统，还有PHP。更多的镜像包括其他的软件也都是可用的，比如Unix和Windows软件。从这个意义上说，AMI本质上与引导磁盘是相同的(尽管我们稍后将返回一些重要的差异)。

一个EC2客户炫耀选择一个配置好的AMI。一个AMI可以启动，从而产生所谓的**EC2实例**：即可用于承载客户应用程序的实际虚拟机。一个重要的话题是客户很难知道一个实例确切是在哪里被运行的。明显的，他是在一个物理机上，但是是哪里的物理机却不得而知。客户能够知道的最确切的机器定位是由亚马逊提供的在哪个区域（美国，南美，欧洲，亚洲等）。

为了方便通信，每个实例都有两个IP地址：一个私有IP支持了内网访问，一个公网IP支持互联网客户的访问。这个共有IP和私有IP的映射使用的是**Network Address Translation（NAT）技术**。一个管理实例的简单方法是使用SSH连接，Amazon负责提供合适的私钥的生成。

提供实例运行环境的EC2提供了几种不同的服务：

**CPU**：允许选择不同数量个种类的CPU，包括GPU

**内存**：决定实例分配多少主存

**存储**：决定分配多少存储空间

**平台**：32位还是64位

**网络**：分配带宽

另外，其他的资源比如网络接口也可以申请。本地存储不是持久的：当实例终止时，所有的数据就会丢失。为了阻止数据丢失，用户需要将数据持久化，比如，使用亚马逊的Simple Storage Service（S3）。另一种方法是附加一个映射到Amazon **Elastic Block Store (Amazon EBS)** 的存储设备。同样，这是另一种服务，但它可以以虚拟块设备的形式使用，而虚拟块设备只是像挂载额外的硬盘一样挂载。当实例停止时，一个EBS设备也可以被挂载到任何其他的实例上。

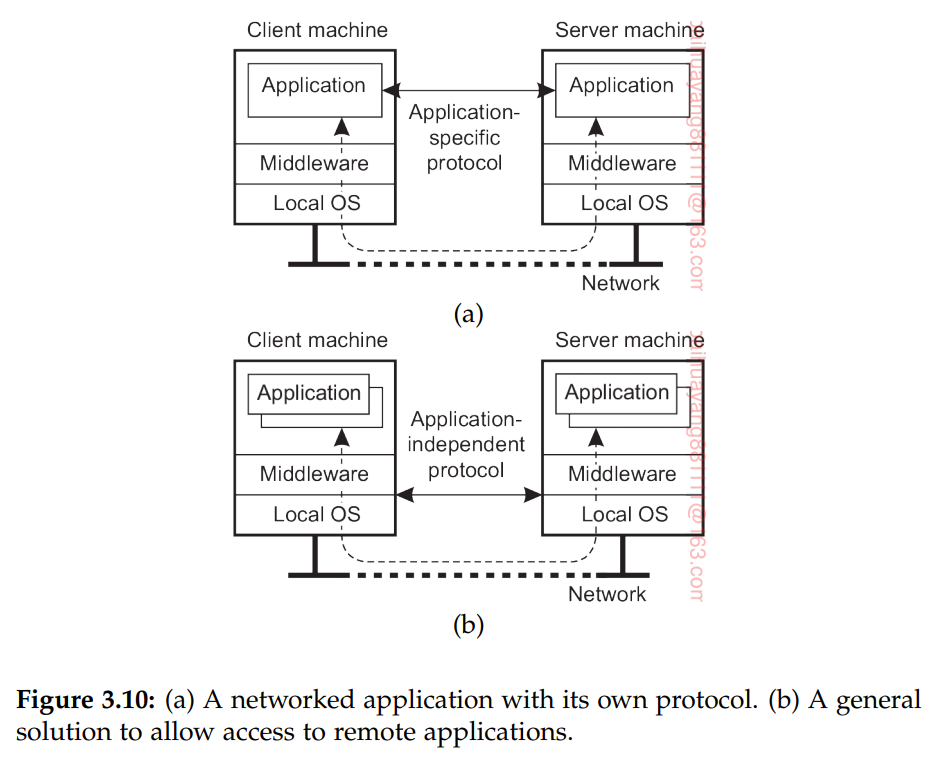
现在应该很清楚了，在没有深入任何重要细节的情况下，EC2提供的IaaS允许客户创建(可能很大)数量的虚拟机，每个虚拟机都根据需要配置了资源，并且能够通过IP网络交换消息。另外，这些虚拟机可以被网络上的可信用户访问。因此，Amazon EC2和许多其他IaaS提供者一样，提供了配置完整分布式系统的方法，该系统由网络虚拟服务器和运行客户提供的分布式应用程序组成。与此同时，这些客户将不需要维护任何物理机器，而这通常已经是一个巨大的收益，我们将在本文中多次遇到这种情况。实际上可以说虚拟化在云计算中起到了核心的作用。

## 3.3客户端

在前几章中，我们讨论了客户机-服务器模型、客户机和服务器的角色以及它们之间的交互方式。现在我们来近距离分别解剖这种客户-服务端的模型。我们这章讨论客户端，下一章讨论服务端。

**网络用户接口**

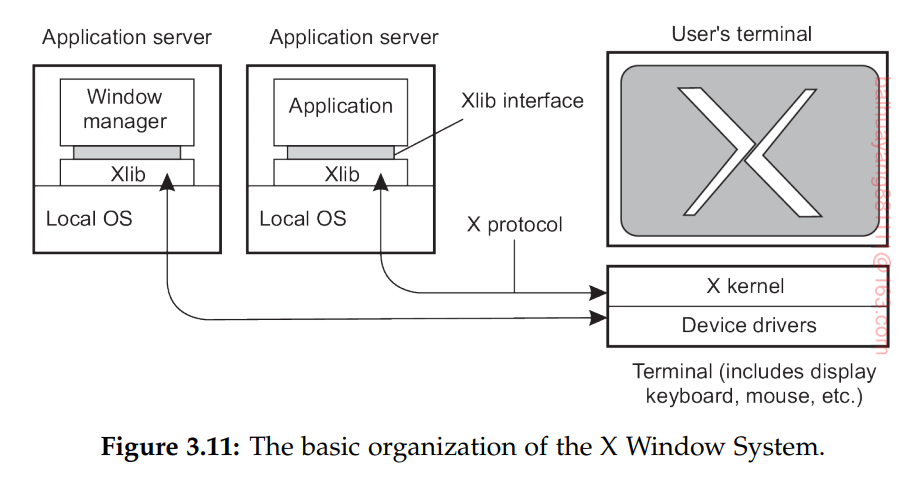
对于客户端一个重要的任务是提供访问远程服务器的手段。这里有两种方式可以支持。第一，首先，对于每个远程服务，客户机机器都有一个单独的对等方，可以通过网络与服务联系。一个典型的栗子是，用户智能手机的日历系统需要同服务端的同步。在这种情况下，应用级别的协议将会处理这种同步，如图3.10(a)所示。



第二个解决方案是通过只提供方便的用户界面来提供对远程服务的直接访问。实际上，这意味着客户机仅作为终端使用，不需要本地存储，从而得到与应用程序无关的解决方案，如图3.10(b)所示。在这种用户网络接口中，所有事物都是存储在服务端的。随着网络连接的增加和移动设备的使用，这种 thin-client approach 已经受到了更多的关注。Thin-client 解决方法同样因为使管理系统任务变得容易而有名。

**例子：the X window system**

比如一个很老但使用很广泛的网络用户接口的例子是**X window system**。X窗口系统，通常简称为X，用于控制位图终端，包括显示器、键盘和鼠标等指向设备。除了支持台式电脑和工作站等传统终端，X还支持平板电脑和智能手机上的触摸屏等现代设备。在某种意义上，X可以看作是控制终端的操作系统的一部分。系统的核心是我们提到的**X kernel**。它包含所有特定于终端的设备驱动程序，因此通常高度依赖于硬件。X内核提供了一个相对底层的接口来控制屏幕，还可以从键盘和鼠标捕获事件。这个接口作为一个名为Xlib的库提供给应用程序。这个通用的组织在图3.11中显示。注意，应用程序很少直接使用Xlib，相反，应用程序更容易部署使用在Xlib上实现的工具包。



X的有趣之处在于，X内核和X应用程序不一定要驻留在同一台机器上。特别是，X提供了X协议，这是一个应用程序级的通信协议，通过它，Xlib的实例可以与X内核交换数据和事件。例如，Xlib可以向X内核发送创建或关闭窗口、设置颜色和定义要显示的光标类型的请求，以及其他许多请求。反过来，X内核将通过向Xlib发送事件包来响应本地事件，如键盘和鼠标输入。

若干个应用可以同时和X kernel 通信。这里有一个特殊的应用有着特殊的权限，我们称之为**window manager**。这个应用程序可以在显示给用户时指定显示的“外观和感觉”。例如，窗口管理器可以规定如何用额外的按钮装饰每个窗口，如何将窗口放置在显示器上，等等。其他应用程序必须遵守这些规则。实际上，这意味着应用程序和X终端之间的大部分交互都是通过窗口管理器重定向的。

值得注意的是，X window系统实际上是如何适应client-server计算的。根据我们到目前为止所描述的，应该很清楚，X内核接收到操作显示的请求。它从(可能是远程)应用程序获取这些请求。从这个意义上说，X内核充当服务器，而应用程序扮演客户机的角色。这个术语已经被X采用，虽然严格来说是正确的，但它很容易导致混淆。