- # Virtualization

Threads and processes can be seen as a way to do more things at the same time. In effect, they allow us to build (pieces of) programs that appear to be executed simultaneously. On a single-processor computer, this simultaneous execution is, of course, an illusion. As there is only a single CPU, only an instruction from a single thread or process will be executed at a time. By rapidly switching between threads and processes, the illusion of parallelism is created.

This separation between having a single CPU and being able to pretend there are more can be extended to other resources as well, leading to what is known as resource virtualization. This virtualization has been applied for many decades, but has received renewed interest as (distributed) computer systems have become more commonplace and complex, leading to the situation that application software is mostly always outliving its underlying systems software and hardware.

The Role of Virtualization in Distributed Systems

In practice, every (distributed) computer system offers a programming interface to higher level software, as shown in Fig. 3-5(a). There are many different types of interfaces, ranging from the basic instruction set as offered by a CPU to the vast collection of application programming interfaces that are shipped with many current middleware systems. In its essence, virtualization deals with extending or replacing an existing interface so as to mimic the behavior of another system, as shown in Fig.3-5(b). We will come to discuss technical details on virtualization shortly, but let us first concentrate on why virtualization is important for distributed systems.

One of the most important reasons for introducing virtualization in the 1970s, was to allow legacy software to run on expensive mainframe hardware. The software not only included various applications, but in fact also the operating systems they were developed for. This approach toward supporting legacy software has been successfully applied on the IBM 370 mainframes (and their successors) that offered a virtual machine to which different operating systems had been ported.

As hardware became cheaper, computers became more powerful, and the number of different operating system flavors was reduced, virtualization became less of an issue.

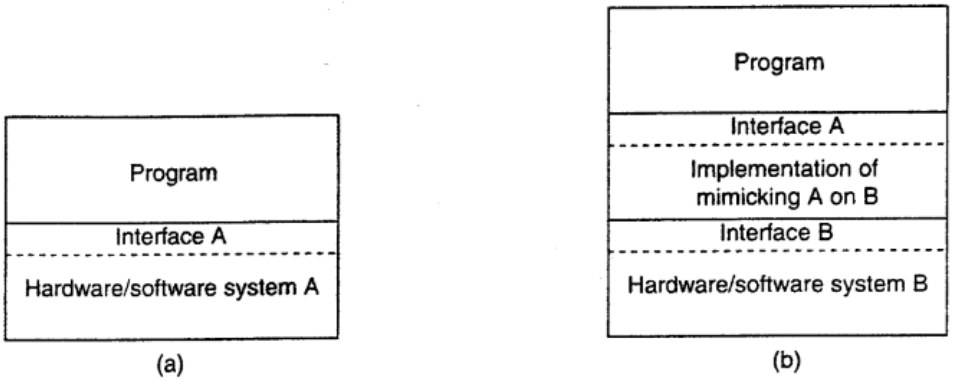

Figure 3-5. (a) General organization between a program, interface, and system. (b) General organization of virtualizing system A on top of system B.

First, while hardware and low-level systems software change reasonably fast, software at higher levels of abstraction (e.g., middleware and applications), are much more stable. In other words, we are facing the situation that legacy software cannot be maintained at the same pace as the platforms it relies on. Virtualization can help here by porting the legacy interfaces to the new platforms and thus immediately opening up the latter for large classes of existing programs.

Equally important is the fact that networking has become completely pervasive. It is hard to imagine that a modern computer is not connected to a network. In practice, this connectivity requires that system administrators maintain a large and heterogeneous collection of server computers, each one running very different applications, which can be accessed by clients. At the same time the various resources should be easily accessible to these applications. Virtualization can help a lot: the diversity of platforms and machines can be reduced by essentially letting each application run on its own virtual machine, possibly including the related libraries and operating system, which, in turn, run on a common platform.

This last type of virtualization provides a high degree of portability and flexibility. For example, in order to realize content delivery networks that can easily support replication of dynamic content, Awadallah and Rosenblum (2002) argue that management becomes much easier if edge servers would support virtualization, allowing a complete site, including its environment to be dynamically copied. As we will discuss later, it is primarily such portability arguments that make virtualization an important mechanism for

distributed systems.

Architectures of Virtual Machines

There are many different ways in which virtualization can be realized in practice. An overview of these various approaches is described by Smith and Nair (2005). To understand the differences in virtualization, it is important to realize that computer systems generally offer four different types of interfaces, at four different levels:

1. An interface between the hardware and software, consisting of machine instructions that can be invoked by any program.

2. An interface between the hardware and software, consisting of machine instructions that can be invoked only by privileged programs, such as an operating system.

3. An interface consisting of system calls as offered by an operating system.

4. An interface consisting of library calls, generally forming what is known as an application programming interface (API). In many cases, the aforementioned system calls are hidden by an API.

These different types are shown in Fig. 3-6. The essence of virtualization is to mimic the behavior of these interfaces.

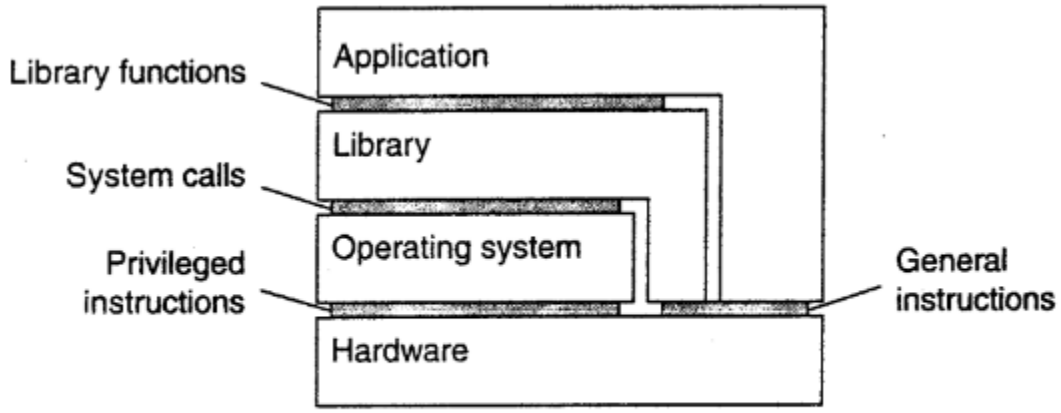

Figure 3-6. Various interfaces offered by computer systems.

Virtualization can take place in two different ways. First, we can build a runtime system that essentially provides an abstract instruction set that is to be used for executing applications. Instructions can be interpreted (as is the case for the

Java runtime environment), but could also be emulated as is done for running Windows applications on UNIX platforms. Note that in the latter case, the emulator will also have to mimic the behavior of system calls, which has proven to be generally far from trivial. This type of virtualization leads to what Smith and Nair (2005) call a process virtual machine, stressing that virtualization is done essentially only for a single process.

An alternative approach toward virtualization is to provide a system that is essentially implemented as a layer completely shielding the original hardware, but offering the complete instruction set of that same (or other hardware) as an interface. Crucial is the fact that this interface can be offered simultaneously to different programs. As a result, it is now possible to have multiple, and different operating systems run independently and concurrently on the same platform. The layer is generally referred to as a virtual machine monitor (VMM). Typical examples of this approach are VMware (Sugerman et al., 200I) and Xen (Barham et at, 2003). These two different approaches are shown in Fig. 3-7.

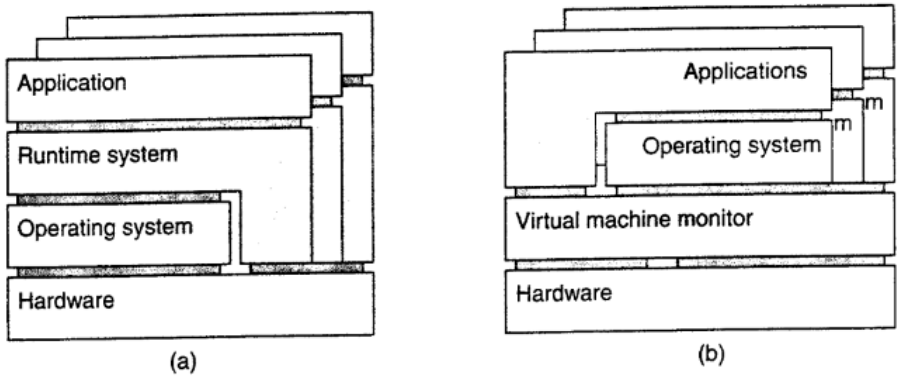

Figure 3-7. (a) A process virtual machine, with multiple instances of (application, runtime) combinations. (b) A virtual machine monitor, with multiple instances of (applications, operating system) combinations.