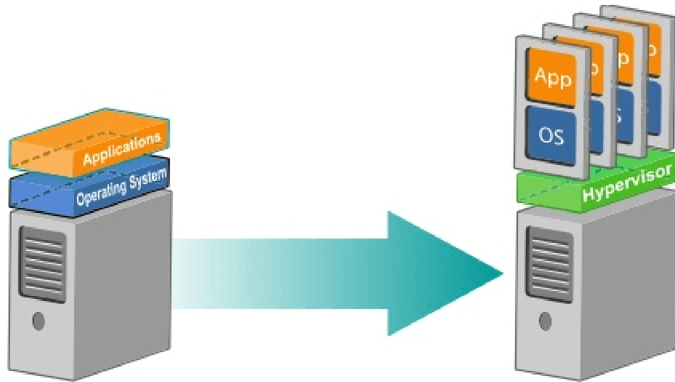
UNIT-IV

VIRTUALIZATION

Virtualization is a kind of technology that is rapidly transforming the IT landscape and has changed the way people compute. It reduces hardware utilization, saves energy and costs and makes it possible to run multiple applications and various operating systems on the same SERVER at the same time. It increases the utilization, efficiency and flexibility of existing computer hardware.



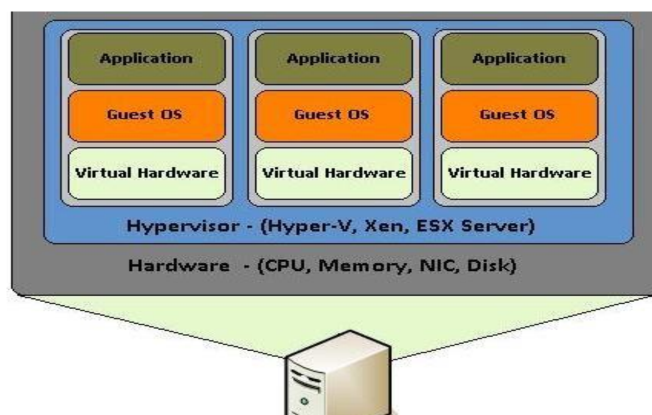
1. NEED FOR VIRTUALIZATION

Virtualization provides various benefits including saving time and energy, decreasing costs and minimizing overall risk.

- Provides ability to manage resources effectively.
- Increases productivity, as it provides secure remote access.
- Provides for data loss prevention.

What makes virtualization possible?

There is a software that makes virtualization possible. This software is known as a Hypervisor, also known as a virtualization manager. It sits between the hardware and the operating system, and assigns the amount of access that the applications and operating systems have with the processor and other hardware resources.



2. TYPES OF VIRTUALIZATION

It would be easier to understand virtualization once we know about different types of virtualization, which are as follows –

Virtualization

Hardware	Network	Storage	Memory	Software	Data	Desktop
• Full • Bare-Metal • Hosted • Partial • Para	• Internal Network Virtualization • External Network Virtualization	• Block Virtualization • File Virtualization	• Application Level Integration • OS Level Integration	• OS Level • Application • Service	• Database	• Virtual desktop infrastructure • Hosted Virtual Desktop

Let's take them one by one.

Hardware/Server Virtualization

It is the most common type of virtualization as it provides advantages of hardware utilization and application uptime. The basic idea of the technology is to combine many small physical servers into one large physical server, so that the processor can be used more effectively and efficiently. The operating system that is running on a physical server gets converted into a well-defined OS that runs on the virtual machine.

The hypervisor controls the processor, memory, and other components by allowing different OS to run on the same machine without the need for a source code.

Hardware virtualization is further subdivided into the following types:

- **Full Virtualization** – In it, the complete simulation of the actual hardware takes place to allow software to run an unmodified guest OS.
- **Para Virtualization** – In this type of virtualization, software unmodified runs in modified OS as a separate system.
- **Partial Virtualization** – In this type of hardware virtualization, the software may need modification to run.

Network Virtualization

It refers to the management and monitoring of a computer network as a single managerial entity from a single software-based administrator's console. It is intended to allow network optimization of data transfer rates, scalability, reliability, flexibility, and security. It also automates many network administrative tasks. Network virtualization is specifically useful for networks experiencing a huge, rapid, and unpredictable increase of usage.

The intended result of network virtualization provides improved network productivity and efficiency.

Two categories:

- **Internal:** Provide network like functionality to a single system.
- **External:** Combine many networks, or parts of networks into a virtual unit.

Storage Virtualization

In this type of virtualization, multiple network storage resources are present as a single storage device for easier and more efficient management of these resources. It provides various advantages as follows:

- Improved storage management in a heterogeneous IT environment
- Easy updates, better availability
- Reduced downtime
- Better storage utilization
- Automated management

In general, there are two types of storage virtualization:

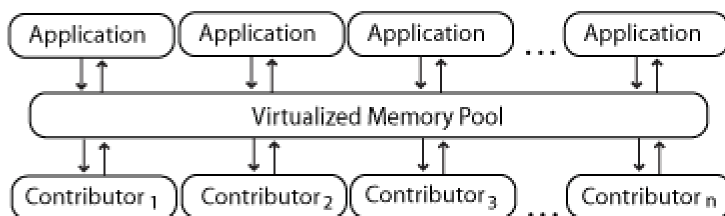
- **Block-** It works before the file system exists. It replaces controllers and takes over at the disk level.
- **File-** The server that uses the storage must have software installed on it in order to enable file-level usage.

Memory Virtualization

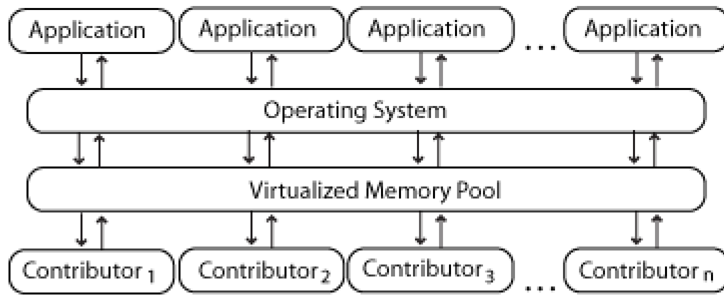
It introduces a way to decouple memory from the server to provide a shared, distributed or networked function. It enhances performance by providing greater memory capacity without any addition to the main memory. That's why a portion of the disk drive serves as an extension of the main memory.

Implementations –

- **Application-level integration** – Applications running on connected computers directly connect to the memory pool through an API or the file system.



- **Operating System Level Integration** – The operating system first connects to the memory pool, and makes that pooled memory available to applications.



Software Virtualization

It provides the ability to the main computer to run and create one or more virtual environments. It is used to enable a complete computer system in order to allow a guest OS to run. For instance letting Linux to run as a guest that is natively running a Microsoft Windows OS (or vice versa, running Windows as a guest on Linux).

Types:

- Operating system
- Application virtualization
- Service virtualization

Data Virtualization

Without any technical details, you can easily manipulate data and know how it is formatted or where it is physically located. It decreases the data errors and workload.

Desktop virtualization

It provides the work convenience and security. As one can access remotely, you are able to work from any location and on any PC. It provides a lot of flexibility for employees to work from home or on the go. It also protects confidential data from being lost or stolen by keeping it safe on central servers.

3. PROS AND CONS OF VIRTUALIZATION

Benefits of virtualization offered up by both organizations as well as vendors:

- **Lower overall capital expenditures.** Virtualization means that you don't need to purchase a server for every single application that you want to implement in your organization. By hosting multiple virtual servers on a single physical machine, you dramatically reduce your cost overhead.

- **Automated tasks**. Virtualization lets you automate a number of significant routine IT tasks. Something as simple as Operating System patches become much simpler and quicker.

- **Greater redundancy.** Virtualization should improve your uptime. Virtualization technologies allow greater safety and security while reducing the points of contact.

- **Faster deployment.** In an virtualized environment, provisioning becomes quick and simple. Deploying a virtual machine is overwhelmingly simpler than deploying a physical machine.

There can be some downsides to virtualization, as well:

- **High upfront expenditures.** When you're implementing a virtualization strategy from the ground up, chances are you're going to have to sink more money into hardware in the immediate future. While you'll save in the long run, implementation can get pricey.

- **Not all applications are ready for virtualization**. There are still vendors not fully supporting virtualized environments.

- **The danger of server sprawl.** Because servers are so easy to deploy in a virtualized environment, there's always the danger that new servers will be added even when they're not needed. Instead of the 10 or 20 virtual servers you really need, you might have 30 or 40.

Pros and Cons of Virtualization

Pros:
Sandbox
Hardware independent
OS independent
Fast Recovery
Live Backup
Migrate data
Reduced Hardware
Run Multiple OS Simultaneously
Cost savings
Use of Multicore processors
System Security
Test and Development

Cons:
Less Efficient
Unstable Performance
Tools lack ability
Rapid Deployment
Latency of Virtual Disk
Backup and Data Sets
Security Issues
Hardware compatibility issues
Managing and Securing is difficult

4. VIRTUAL MACHINE

Virtual Machine is a completely separate individual operating system installation on your usual operating system. It is implemented by software emulation and hardware virtualization.

Virtual machine is a software implementation of a physical machine - computer - that works and executes analogically to it. Virtual machines are divided in two categories based on their use and correspondence to real machine: system virtual machines and process virtual machines. First category provides a complete system platform that executes complete operating system, second one will run a single program.

Frequently multiple virtual machines with their own OS's are used in server consolidation, where different services are run in separate virtual environments, but on the same physical machine.

The main advantages of virtual machines:

- Multiple OS environments can exist simultaneously on the same machine, isolated from each other;
- Virtual machine can offer an instruction set architecture that differs from real computer's;
- Easy maintenance, application provisioning, availability and convenient recovery.

The main disadvantages:

- When multiple virtual machines are simultaneously running on a host computer, each virtual machine may introduce an unstable performance, which depends on the workload on the system by other running virtual machines;
- Virtual machine is not that efficient as a real one when accessing the hardware.

TYPES OF VIRTUAL MACHINE

virtual machines can be divided into two categories:

1. System Virtual Machines: A system platform that supports the sharing of the host computer's physical resources between multiple virtual machines, each running with its own copy of the operating system. The virtualization technique is provided by a software layer known as a hypervisor, which can run either on bare hardware or on top of an operating system.
2. Process Virtual Machine: Designed to provide a platform-independent programming environment that masks the information of the underlying hardware or operating system and allows program execution to take place in the same way on any given platform.

5. PROCESS VIRTUAL MACHINE

1.3 Process Virtual Machines

Process-level VMs provide user applications with a virtual ABI environment. In their various implementations, process VMs can provide replication, emulation, and optimization. The following subsections describe each of these.

1.3.1 Multiprogramming

The first and most common virtual machine is so ubiquitous that we don't even think of it as being a virtual machine. The combination of the OS call interface and the user instruction set forms the machine that executes a user process. Most operating systems can simultaneously support multiple user processes through multiprogramming, where each user process is given the illusion of having a complete machine to itself. Each process is given its own address space and is given access to a file structure. The operating system time-shares the hardware and manages underlying resources to make this possible. In effect, the operating system provides a replicated process-level virtual machine for each of the concurrently executing applications.

1.3.2 Emulators and Dynamic Binary Translators

A more challenging problem for process-level virtual machines is to support program binaries compiled to a different instruction set than the one executed by the host's hardware, i.e., to *emulate* one instruction set on hardware designed for another. An example emulating process virtual machine is illustrated in Figure 1.9. Application programs are compiled for a *source ISA*, but

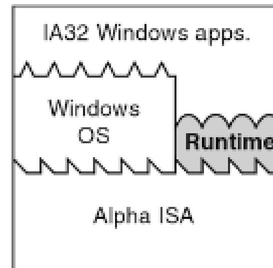


Figure 1.9 A Process VM That Emulates Guest Applications. *The Digital FX!32 system allows Windows IA-32 applications to be run on an Alpha Windows platform.*

the hardware implements a different *target ISA*. As shown in the example, the operating system may be the same for both the guest process and the host platform, although in other cases the operating systems may differ as well. The example in Figure 1.9 illustrates the Digital FX!32 system (Hookway and Herdeg 1997). The FX!32 system can run Intel IA-32 application binaries compiled for Windows NT on an Alpha hardware platform also running Windows NT. More recent examples are the Aries system (Zheng and Thompson 2000) which supports PA-RISC programs on an IPF (Itanium) platform, and the Intel IA-32 EL (execution layer) which supports IA-32 programs on an IPF platform (Baraz et al. 2003).

The most straightforward emulation method is *interpretation*. An interpreter program executing the target ISA fetches, decodes, and emulates the execution of individual source instructions. This can be a relatively slow process, requiring tens of native target instructions for each source instruction interpreted.

For better performance, *binary translation* is typically used. With binary translation, blocks of source instructions are converted to target instructions that perform equivalent functions. There can be a relatively high overhead associated with the translation process, but once a block of instructions is translated, the translated instructions can be cached and repeatedly executed much faster than they can be interpreted. Because binary translation is the most important feature of this type of process virtual machine, they are sometimes called *dynamic binary translators*.

Interpretation and binary translation have different performance characteristics. Interpretation has relatively low startup overhead but consumes significant time whenever an instruction is emulated. On the other hand, binary translation has high initial overhead when performing the translations but is fast for each repeated execution. Consequently, some virtual machines use a staged emulation strategy combined with *profiling*, i.e., the collection of

statistics regarding the program's behavior. Initially, a block of source instructions is interpreted, and profiling is used to determine which instruction sequences are frequently executed. Then a frequently executed block may be binary translated. Some systems perform additional code optimizations on the translated code if profiling shows that it has a very high execution frequency. In most emulating virtual machines the stages of interpretation and binary translation can both occur over the course of a single program's execution. In the case of FX!32, translation occurs incrementally between program runs.

1.3.3 Same-ISA Binary Optimizers

Most dynamic binary translators not only translate from source to target code but also perform some code optimizations. This leads naturally to virtual machines where the instruction sets used by the host and the guest are the same, and optimization of a program binary is the primary purpose of the virtual machine. Thus, *same-ISA dynamic binary optimizers* are implemented in a manner very similar to emulating virtual machines, including staged optimization and software caching of optimized code. Same-ISA dynamic binary optimizers are most effective for source binaries that are relatively unoptimized to begin with, a situation that is fairly common in practice. A dynamic binary optimizer can collect a profile and then use this profile information to optimize the binary code on the fly. An example of such a same-ISA dynamic binary optimizer is the Dynamo system, originally developed as a research project at Hewlett-Packard (Bala, Duesterwald, and Banerjia 2000).

1.3.4 High-Level Language Virtual Machines: Platform Independence

For the process VMs described earlier, cross-platform portability is clearly a very important objective. For example, the FX!32 system enabled portability of application software compiled for a popular platform (IA-32 PC) to a less popular platform (Alpha). However, this approach allows cross-platform compatibility only on a case-by-case basis and requires a great deal of programming effort. For example, if one wanted to run IA-32 binaries on a number of hardware platforms currently in use, e.g., SPARC, PowerPC, and MIPS, then an FX!32-like VM would have to be developed for each of them. The problem would be even more difficult if the host platforms run different operating systems than the one for which binaries were originally compiled.

Full cross-platform portability is more easily achieved by taking a step back and designing it into an overall software framework. One way of accomplishing

this is to design a process-level VM at the same time as an application development environment is being defined. Here, the VM environment does not directly correspond to any real platform. Rather, it is designed for ease of portability and to match the features of a high-level language (HLL) used for application program development. These *high-level language VMs* (HLL VMs) are similar to the process VMs described earlier. However, they are focused on minimizing hardware-specific and OS-specific features because these would compromise platform independence.

High-level language VMs first became popular with the Pascal programming environment (Bowles 1980). In a conventional system, Figure 1.10a, the compiler consists of a frontend that performs lexical, syntax, and semantic analysis to generate simple intermediate code — similar to machine code but more abstract. Typically the intermediate code does not contain specific register assignments, for example. Then a code generator takes the intermediate code and generates a binary containing machine code for a specific ISA and OS. This binary file is distributed and executed on platforms that support the given ISA/OS combination. To execute the program on a different platform, however, it must be recompiled for that platform.

In HLL VMs, this model is changed (Figure 1.10b). The steps are similar to the conventional ones, but the point at which program distribution takes place is at a higher level. As shown in Figure 1.10b, a conventional compiler frontend generates abstract machine code, which is very similar to an intermediate form. In many HLL VMs, this is a rather generic stack-based ISA. This *virtual ISA*

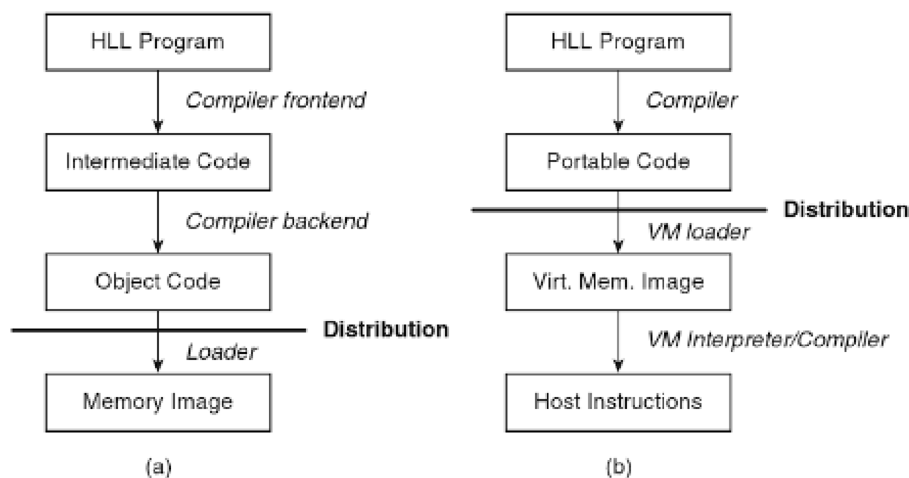


Figure 1.10 High-Level Language Environments. *(a) A conventional system, where platform-dependent object code is distributed; (b) an HLL VM environment, where portable intermediate code is "executed" by a platform-dependent virtual machine.*

is in essence the machine code for a virtual machine. The portable virtual ISA code is distributed for execution on different platforms. For each platform, a VM capable of executing the virtual ISA is implemented. In its simplest form, the VM contains an interpreter that takes each instruction, decodes it, and then performs the required state transformations (e.g., involving memory and the stack). I/O functions are performed via a set of standard library calls that are defined as part of the VM. In more sophisticated, higher performance VMs, the abstract machine code may be compiled (binary translated) into host machine code for direct execution on the host platform.

An advantage of an HLL VM is that software is easily portable, once the VM is implemented on a target platform. While the VM implementation would take some effort, it is a much simpler task than developing a compiler for each platform and recompiling every application when it is ported. It is also simpler than developing a conventional emulating process VM for a typical real-world ISA.

6. SYSTEM VIRTUAL MACHINE

1.4 System Virtual Machines

System virtual machines provide a complete system environment in which many processes, possibly belonging to multiple users, can coexist. These VMs were first developed during the 1960s and early 1970s, and they were the origin of the term *virtual machine*. By using system VMs, a single host hardware platform can support multiple guest OS environments simultaneously. At the time they were first developed, mainframe computer systems were very large and expensive, and computers were almost always shared among a large number of users. Different groups of users sometimes wanted different operating systems

to be run on the shared hardware, and VMs allowed them to do so. Alternatively, a multiplicity of single-user operating systems allowed a convenient way of implementing time-sharing among many users. Over time, as hardware became much less expensive and much of it migrated to the desktop, interest in these classic system VMs faded.

Today, however, system VMs are enjoying renewed popularity. This is partly due to modern-day variations of the traditional motivations for system VMs. The large, expensive mainframe systems of the past are now servers or server farms, and these servers may be shared by a number of users or user groups. Perhaps the most important feature of today's system VMs is that they provide a secure way of partitioning major software systems that run concurrently on the same hardware platform. Software running on one guest system is isolated from software running on other guest systems. Furthermore, if security on one guest system is compromised or if the guest OS suffers a failure, the software running on other guest systems is not affected. The ability to support different operating systems simultaneously, e.g., Windows and Linux (as illustrated in Figure 1.11), is another reason for their appeal, although it is probably of secondary importance to most users.

In system VMs, platform replication is the major feature provided by a VMM. The central problem is that of dividing a single set of hardware resources among multiple guest operating system environments. The VMM has access to and manages all the hardware resources. A guest operating system and application programs compiled for that operating system are then managed under (hidden) control of the VMM. This is accomplished by constructing the system so that when a guest OS performs certain operations, such as a privileged instruction that directly involves the shared hardware resources, the operation is intercepted by the VMM, checked for correctness, and performed by the VMM on behalf of the guest. Guest software is unaware of the "behind-the-scenes" work performed by the VMM.

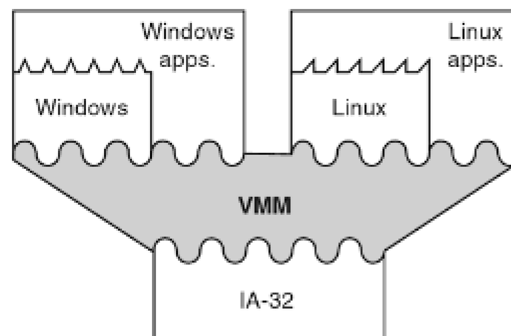


Figure 1.11 A System VM That Supports Multiple OS Environments on the Same Hardware.

1.4.1 Implementations of System Virtual Machines

From the user perspective, most system VMs provide more or less the same functionality. The thing that tends to differentiate them is the way in which they are implemented. As discussed earlier, in Section 1.2, there are a number of interfaces in a computer system, and this leads to a number of choices for locating the system VMM software. Summaries of two of the more important implementations follow.

Figure 1.11 illustrates the classic approach to system VM architecture (Popek and Goldberg 1974). The VMM is first placed on bare hardware, and virtual machines fit on top. The VMM runs in the most highly privileged mode, while all the guests systems run with lesser privileges. Then in a completely transparent way, the VMM can intercept and implement all the guest OS's actions that interact with hardware resources. In many respects, this system VM architecture is the most efficient, and it provides service to all the guest systems in a more or less equivalent way. One disadvantage of this type of system, at least for desktop users, is that installation requires wiping an existing system clean and starting from scratch, first installing the VMM and then installing guest operating systems on top. Another disadvantage is that I/O device drivers must be available for installation in the VMM, because it is the VMM that interacts directly with I/O devices.

An alternative system VMM implementation builds virtualizing software on top of an existing host operating system — resulting in what is called a *hosted VM*. With a hosted VM, the installation process is similar to installing a typical application program. Furthermore, virtualizing software can rely on the host OS to provide device drivers and other lower-level services; they don't have to be provided by the VMM. The disadvantage of this approach is that there can be some loss of efficiency because more layers of software become involved when OS service is required. The hosted VM approach is taken in the VMware implementation (VMware 2000), a modern system VM that runs on IA-32 hardware platforms.

1.4.2 Whole System VMs: Emulation

In the conventional system VMs described earlier, all the system software (both guest and host) and application software use the same ISA as the underlying hardware. In some important situations, however, the host and guest systems do not have a common ISA. For example, the Apple PowerPC-based systems and Windows PCs use different ISAs (and different operating systems), and they are the two most popular desktop systems today. As another example,

Sun Microsystems servers use a different OS and ISA than the Windows PCs that are commonly attached to them as clients. Because software systems are so closely tied to hardware systems, this may require purchase of multiple platform types, even when unnecessary for any other reason, which complicates software support and/or restricts the availability of useful software packages to users.

This situation motivates system VMs, where a complete software system, both OS and applications, is supported on a host system that runs a different ISA and OS. These are called *whole-system VMs* because they essentially virtualize all software. Because the ISAs are different, both application and OS code require emulation, e.g., via binary translation. For whole-system VMs, the most common implementation method is to place the VMM and guest software on top of a conventional host OS running on the hardware.

Figure 1.12 illustrates a whole-system VM built on top of a conventional system with its own OS and application programs. An example of this type of VM is Virtual PC (Traut 1997), which enables a Windows system to run on a Macintosh platform. The VM software executes as an application program supported by the host OS and uses no system ISA operations. It is as if the VM software, the guest OS, and guest application(s) are one very large application implemented on the host OS and hardware. Meanwhile the host OS can also continue to run applications compiled for the native ISA; this feature is illustrated in the right-hand section of the drawing.

To implement a system VM of this type, the VM software must emulate the entire hardware environment. It must control the emulation of all the

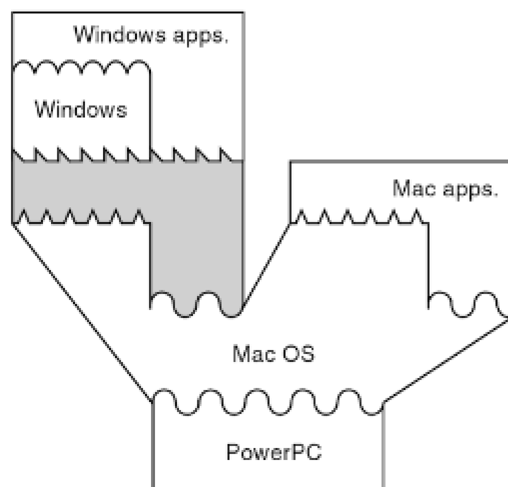


Figure 1.12 A Whole-System VM That Supports a Guest OS and Applications, in Addition to Host Applications.

instructions, and it must convert the guest system ISA operations to equivalent OS calls made to the host OS. Even if binary translation is used, it is tightly constrained because translated code often cannot take advantage of underlying system ISA features such as virtual memory management and trap handling. In addition, problems can arise if the properties of hardware resources are significantly different in the host and the guest. Solving these mismatches is the major challenge of implementing whole-system VMs.

1.4.3 Codesigned Virtual Machines: Hardware Optimization

In all the VM models discussed thus far, the goal has been functionality and portability — either to support multiple (possibly different) operating systems on the same host platform or to support different ISAs and operating systems on the same platform. In practice, these virtual machines are implemented on hardware already developed for some standard ISA and for which native (host) applications, libraries, and operating systems already exist. By and large, improved performance (i.e., going beyond native platform performance) has not been a goal — in fact minimizing performance losses is often the performance goal.

Codesigned VMs have a different objective and take a different approach. These VMs are designed to enable innovative ISAs and/or hardware implementations for improved performance, power efficiency, or both. The host's ISA may be completely new, or it may be based on an existing ISA with some new instructions added and/or some instructions deleted. In a codesigned VM, there are no native ISA applications. It is as if the VM software is, in fact, part of the hardware implementation.

In some respects, codesigned virtual machines are similar to a purely hardware virtualization approach used in many high-performance superscalar microprocessors. In these designs, hardware renames architected registers to a larger number of physical registers, and complex instructions are decomposed into simpler, RISC-like instructions (Hennessy and Patterson 2002). In this book, however, we focus on software-implemented codesigned virtual machines; binary translation is over a larger scope and can be more flexible because it is done in software.

1.5 A Taxonomy

We have just described a rather broad array of VMs, with different goals and implementations. To put them in perspective and organize the common implementation issues, we introduce a taxonomy illustrated in Figure 1.13. First, VMs are divided into the two major types: process VMs and system VMs. In the first type, the VM supports an ABI — user instructions plus system calls; in the second, the VM supports a complete ISA — both user and system instructions. Finer divisions in the taxonomy are based on whether the guest and host use the same ISA.

On the left-hand side of Figure 1.13 are process VMs. These include VMs where the host and guest instruction sets are the same. In the figure, we identify two examples. The first is multiprogrammed systems, as already supported on most of today's systems. The second is same-ISA dynamic binary optimizers, which transform guest instructions only by optimizing them and then execute them natively.

For process VMs where the guest and host ISAs are different, we also give two examples. These are dynamic translators and HLL VMs. HLL VMs are

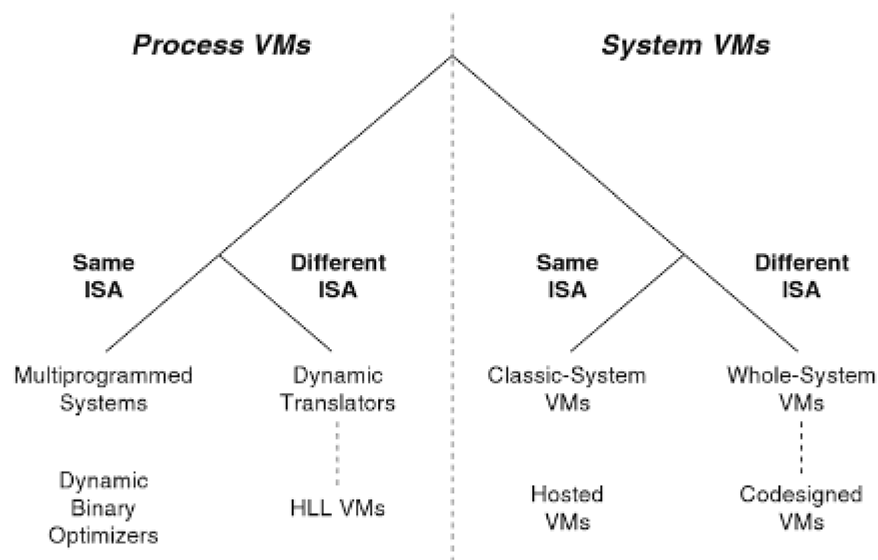


Figure 1.13 A Taxonomy of Virtual Machines.

connected to the VM taxonomy via a "dotted line" because their process-level interface is at a different, higher level than the other process VMs.

On the right-hand side of the figure are system VMs. If the guest and host use the same ISA, examples include "classic" system VMs and hosted VMs. In these VMs, the objective is providing replicated, isolated system environments. The primary difference between classic and hosted VMs is the VMM implementation rather than the function provided to the user.

Examples of system VMs where the guest and host ISAs are different include whole-system VMs and codesigned VMs. With whole-system VMs, performance is often of secondary importance compared to accurate functionality, while with codesigned VMs, performance (and power efficiency) are often major goals. In the figure, codesigned VMs are connected using dotted lines because their interface is typically at a lower level than other system VMs.

7. BINARY TRANSLATION

Depending on implementation technologies, hardware virtualization can be classified into two categories:full virtualization and host-based virtualization. Full virtualization does not need to modify the host OS. It relies on binary translation to trap and to virtualize the execution of certain sensitive, nonvirtualizable instructions. The guest OSes and their applications consist of noncritical and critical instructions. In a host-based system, both a host OS and a guest OS are used. A virtualization software layer is built between the host OS and guest OS.

Full Virtualization

With full virtualization, noncritical instructions run on the hardware directly while critical instructions are discovered and replaced with traps into the VMM to be emulated by software. Both the hypervisor and VMM approaches are considered full virtualization. Why are only critical instructions trapped into the VMM? This is because binary translation can incur a large performance overhead. Noncritical instructions do not control hardware or threaten the security of the system, but critical instructions do. Therefore, running noncritical instructions on hardware not only can promote efficiency, but also can ensure system security.

Binary Translation of Guest OS Requests Using a VMM

This approach was implemented by VMware and many other software companies. As shown in Figure 3.6, VMware puts the VMM at Ring 0 and the guest OS at Ring 1. The VMM scans the instruction stream and identifies the privileged, control- and behavior-sensitive instructions. When these instructions are identified, they are trapped into the VMM, which emulates the behavior of these instructions. The method used in this emulation is called binary translation. Therefore, full virtualization combines binary translation and direct execution. The guest OS is completely decoupled from the underlying hardware. Consequently, the guest OS is unaware that it is being virtualized. The performance of full virtualization may not be ideal, because it involves binary translation which is rather time-consuming. In particular, the full virtualization of I/O-intensive applications is a really a big challenge. Binary translation employs a code cache to store translated hot instructions to improve performance, but it increases the cost of memory usage. At the time of this writing, the performance of full virtualization on the x86 architecture is typically 80 percent to 97 percent that of the host machine.

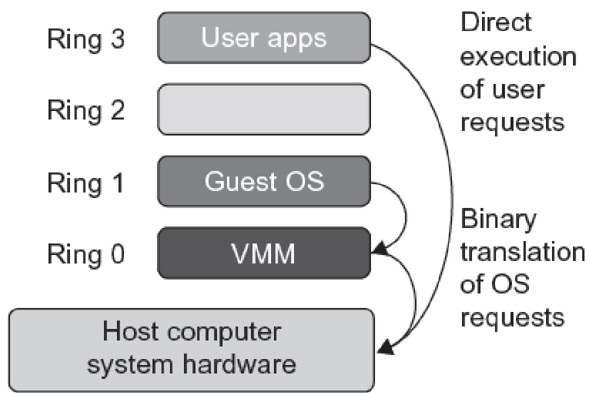


FIGURE 3.6

Indirect execution of complex instructions via binary translation of guest OS requests using the VMM plus direct execution of simple instructions on the same host.

Host-Based Virtualization

An alternative VM architecture is to install a virtualization layer on top of the host OS. This host OS is still responsible for managing the hardware. The guest OSes are installed and run on top of the virtualization layer. Dedicated applications may run on the VMs. Certainly, some other applications can also run with the host OS directly. This host-based architecture has some distinct advantages, as enumerated next. First, the user can install this VM architecture without modifying the host OS. The virtualizing software can rely on the host OS
to provide device drivers and other low-level services. This will simplify the VM design and ease its deployment. Second, the host-based approach appeals to many host machine configurations. Compared to the hypervisor/VMM architecture, the performance of the host-based architecture may also be low. When an application requests hardware access, it involves four layers of mapping which downgrades performance significantly. When the ISA of a guest OS is different from the ISA of the underlying hardware, binary translation must be adopted. Although the host-based architecture has flexibility, the performance is too low to be useful in practice.

Para-Virtualization with Compiler Support

Para-virtualization needs to modify the guest operating systems. A para-virtualized VM provides special APIs requiring substantial OS modifications in user applications. Performance degradation is a critical issue of a virtualized system. No one wants to use a VM if it is much slower than using a physical machine. The virtualization layer can be inserted at different positions in a machine software stack. However, para-virtualization attempts to reduce the virtualization overhead, and thus improve performance by modifying only the guest OS kernel.

Figure 3.7 illustrates the concept of a para-virtualized VM architecture. The guest operating systems are para-virtualized. They are assisted by an intelligent compiler to replace the nonvirtualizable OS instructions by hypercalls as illustrated in Figure 3.8. The traditional x86 processor offers four instruction execution rings: Rings 0, 1, 2, and 3. The lower the ring number, the higher the privilege of instruction being executed. The OS is responsible for managing the hardware and the privileged instructions to execute at Ring 0, while user-level applications run at Ring 3. The best example of para-virtualization is the KVM to be described below.

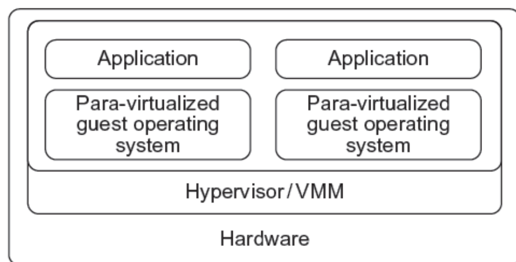


FIGURE 3.7

Para-virtualized VM architecture, which involves modifying the guest OS kernel to replace nonvirtualizable instructions with hypercalls for the hypervisor or the VMM to carry out the virtualization process (See Figure 3.8 for more details.)

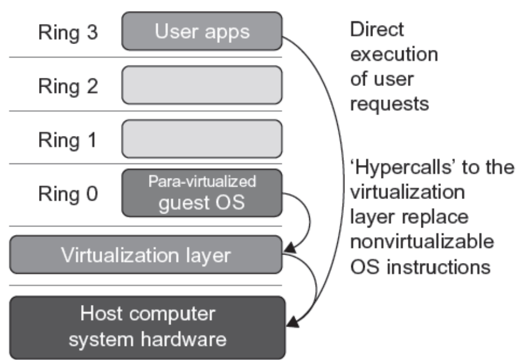


FIGURE 3.8

The use of a para-virtualized guest OS assisted by an intelligent compiler to replace nonvirtualizable OS instructions by hypercalls.

(*Courtesy of VMWare [71]*)

Para-Virtualization Architecture

When the x86 processor is virtualized, a virtualization layer is inserted between the hardware and the OS. According to the x86 ring definition, the virtualization layer should also be installed at Ring 0. Different instructions at Ring 0 may cause some problems. In Figure 3.8, we show that para-virtualization replaces nonvirtualizable instructions with hypercalls that communicate directly with the hypervisor or VMM. However, when the guest OS kernel is modified for virtualization, it can no longer run on the hardware directly.

Although para-virtualization reduces the overhead, it has incurred other problems. First, its compatibility and portability may be in doubt, because it must support the unmodified OS as well. Second, the cost of maintaining para-virtualized OSes is high, because they may require deep OS kernel modifications. Finally, the performance advantage of para-virtualization varies greatly due to workload variations. Compared with full virtualization, para-virtualization is relatively easy and more practical. The main problem in full virtualization is its low performance in binary translation. To speed up binary translation is difficult. Therefore, many virtualization products employ the para-virtualization architecture. The popular Xen, KVM, and VMware ESX are good examples.

KVM (Kernel-Based VM)

This is a Linux para-virtualization system—a part of the Linux version 2.6.20 kernel. Memory management and scheduling activities are carried out by the existing Linux kernel. The KVM does the rest, which makes it simpler than the hypervisor that controls the entire machine. KVM is a hardware-assisted para-virtualization tool, which improves performance and supports unmodified guest OSes such as Windows, Linux, Solaris, and other UNIX variants.

Para-Virtualization with Compiler Support

Unlike the full virtualization architecture which intercepts and emulates privileged and sensitive instructions at runtime, para-virtualization handles these instructions at compile time. The guest OS kernel is modified to replace the privileged and sensitive instructions with hypercalls to the hypervisor or VMM. Xen assumes such a para-virtualization architecture.

The guest OS running in a guest domain may run at Ring 1 instead of at Ring 0. This implies that the guest OS may not be able to execute some privileged and sensitive instructions. The privileged instructions are implemented by hypercalls to the hypervisor. After replacing the instructions with hypercalls, the modified guest OS emulates the behavior of the original guest OS. On an UNIX

system, a system call involves an interrupt or service routine. The hypercalls apply a dedicated service routine in Xen.

8. HIGH LEVEL LANGUAGE VM

Designing a special guest ISA/system interface is known to be HLL VM:

- With portability as the main goal
- Define an abstract interface that can be supported by all conventional OSes.
- Reflects important features of specific HLL or class of HLLs.
- Simplifies compilation

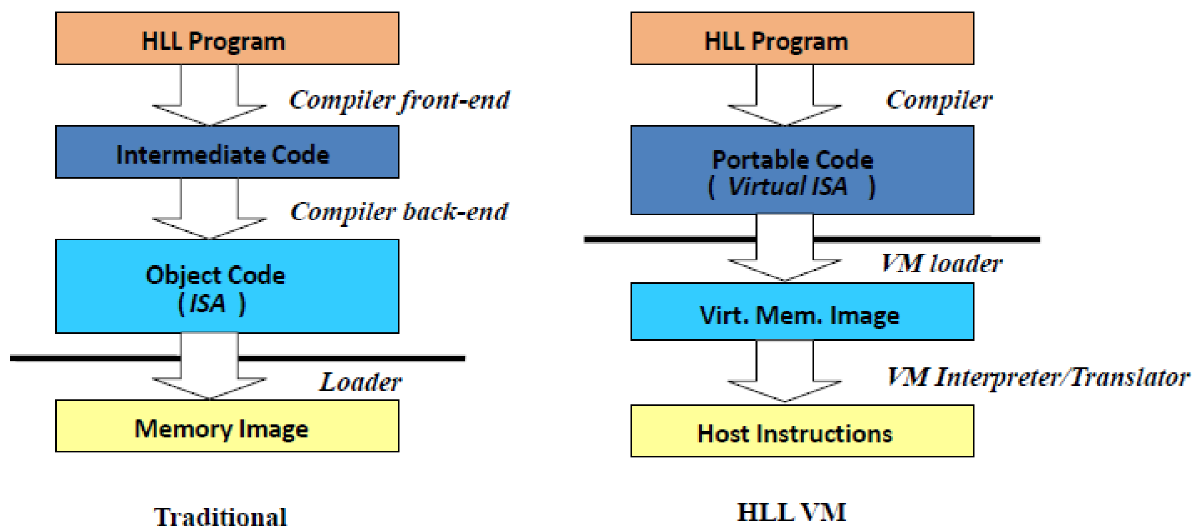
HLL VM is similar to process VM but..

- ISA defined for user-mode programs only
- ISA not designed for real hardware
  - Only to be executed on virtual processor
  - Referred to as virtual-ISA or v-ISA
- System interface is a set of standardized APIs.

HLL VMs from language / compiler perspective

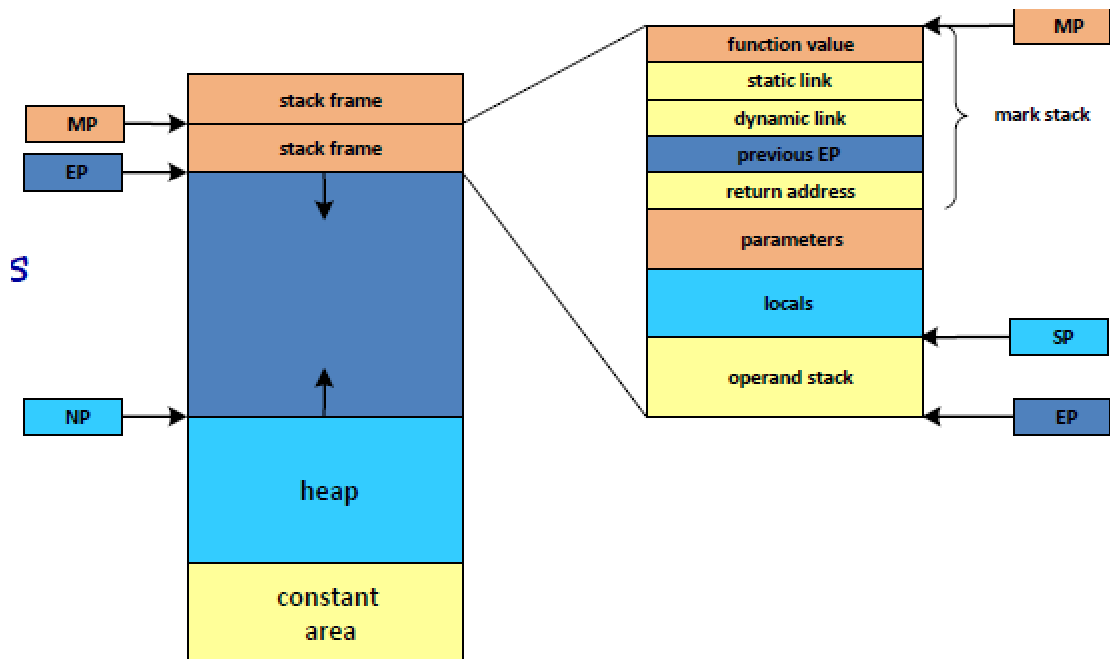
Goal: Complete platform independence for applications

Virtual instruction set + libraries (Instead of ISA and OS interface)



P-Code VM

- Popularized HLL VMs
- Provided highly portable version of Pascal
- Consists of
  - Primitive libraries
  - Machine-independent object file format
  - A set of byte-oriented “pseudo-codes”
  - Virtual machine definition of pseudo-code semantics
- Instruction set
  - Stack oriented
  - Stack “Frame” is part of VM definition



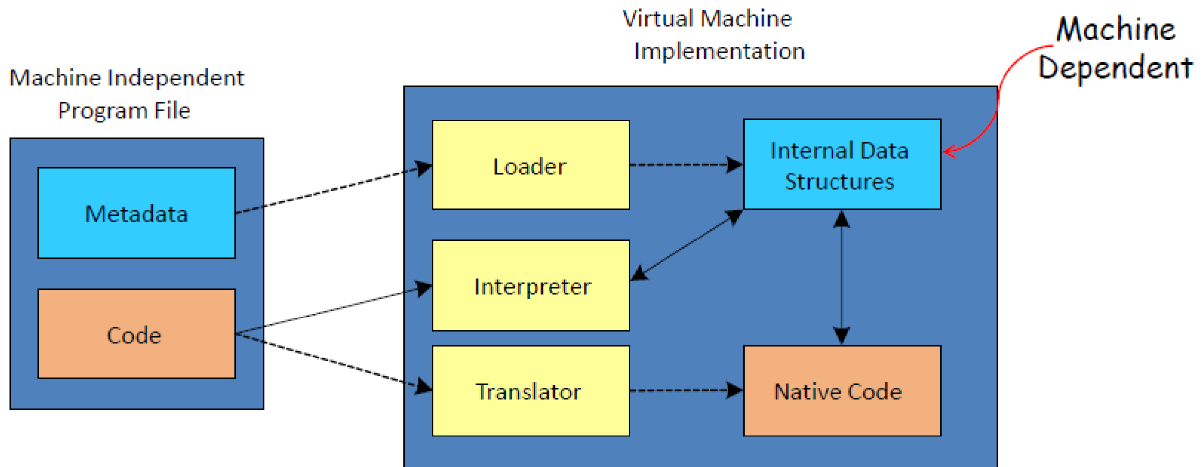
MP-Mark Pointer
EP-Extreme Pointer
NP-New Pointer
SP-Stack Pointer

Advantages

- Porting is simplified
  - Don't have to develop compilers for all platforms
- VM implementation is smaller/simpler than a compiler
- VM provides concise definition of semantics
- Through interpretation, startup time is reduced
- Generic I/O and Memory interface
  - Tended to be least common denominator

Modern HLL VMs

- Superficially similar to P-code scheme
  - Stack-oriented ISA
  - Standard libraries
- Network Computing Environment
  - Untrusted software
  - Robustness
    - Object-oriented programming
  - Bandwidth is a consideration
  - Good performance must be maintained
- Two major examples
  - Java VM
  - Microsoft Common Language Infrastructure (CLI)
- Compiler forms program files (e.g. class files)
  - Standard format
- Program files contain both code and metadata



Java Virtual Machine Architecture --- CLI

- Analogous to an ISA

Java Virtual Machine Implementation -- CLR (Common Language Runtime)

- Analogous to a computer implementation

Java bytecodes – Microsoft Intermediate Language (MSIL), CIL, IL

- The instruction part of the ISA

Java Platform -- .Net framework

- ISA + libraries ; A higher level of ABI

Hypervisor - Xen Architecture

The hypervisor supports hardware-level virtualization (see Figure 3.1(b)) on bare metal devices like CPU, memory, disk and network interfaces. The hypervisor software sits directly between the physical hardware and its OS. This virtualization layer is referred to as either the VMM or the hypervisor. The hypervisor provides hypercalls for the guest OSes and applications. Depending on the functionality, a hypervisor can assume a micro-kernel architecture like the Microsoft Hyper-V. Or it can assume a monolithic hypervisor architecture like the VMware ESX for server virtualization. A micro-kernel hypervisor includes only the basic and unchanging functions (such as physical memory management and processor scheduling). The device drivers and other changeable components are outside the hypervisor. A monolithic hypervisor implements all the aforementioned functions, including those of the device drivers. Therefore, the size of the hypervisor code of a micro-kernel hypervisor is smaller than that of a monolithic hypervisor. Essentially, a hypervisor must be able to convert physical devices into virtual resources dedicated for the deployed VM to use.

9. Hypervisors- The Xen Architecture

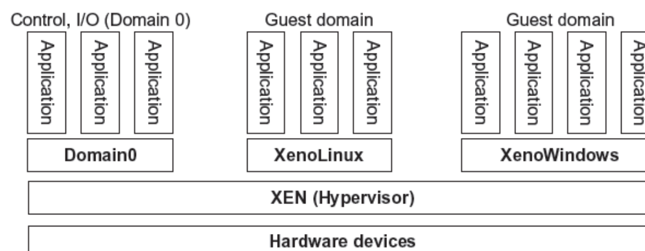


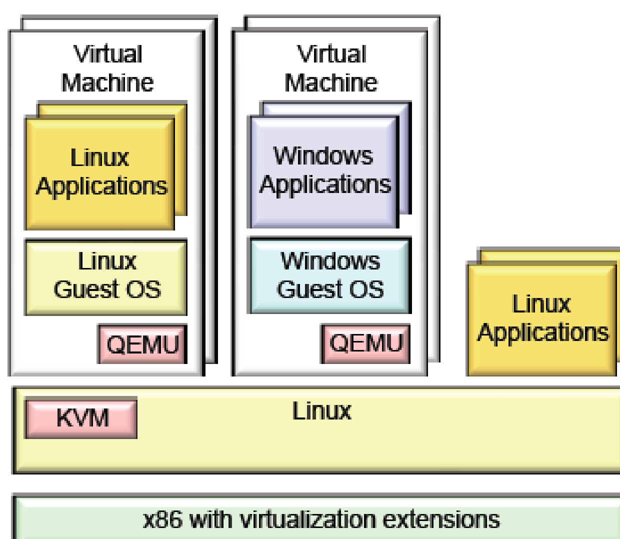
FIGURE 3.5

The Xen architecture's special domain 0 for control and I/O, and several guest domains for user applications.

Xen is an open source hypervisor program developed by Cambridge University. Xen is a microkernel hypervisor, which separates the policy from the mechanism. The Xen hypervisor implements all the mechanisms, leaving the policy to be handled by Domain 0 does not include any device drivers natively It just provides a mechanism by which a guest OS can have direct access to the physical devices. As a result, the size of the Xen hypervisor is kept rather small. Xen provides a virtual environment located between the hardware and the OS. A number of vendors are in the process of developing commercial Xen hypervisors, among them are Citrix XenServer and Oracle VM. The core components of a Xen system are the hypervisor, kernel, and applications. The organization of the three components is important. Like other virtualization systems, many guest OSes can run on top of the hypervisor. However, not all guest OSes are created equal, and one in particular controls the others. The guest OS, which has control ability, is called Domain 0, and the others are called Domain U. Domain 0 is a privileged guest OS of Xen. It is first loaded when Xen boots without any file system drivers being available. Domain 0 is designed to access hardware directly and manage devices. Therefore, one of the responsibilities of Domain 0 is to allocate and map hardware resources for the guest domains (the Domain U domains). For example, Xen is based on Linux and its security level is C2. Its management VM is named Domain 0, which has the privilege to manage other VMs implemented on the same host. If Domain 0 is compromised, the hacker can control the entire system. So, in the VM system, security policies are needed to improve the security of Domain 0. Domain 0, behaving as a VMM, allows users to create, copy, save, read, modify, share, migrate, and roll back VMs as easily as manipulating a file, which flexibly provides tremendous benefits for users. Unfortunately, it also brings a series of security problems during the software life cycle and data lifetime.

Traditionally, a machine's lifetime can be envisioned as a straight line where the current state of the machine is a point that progresses monotonically as the software executes. During this time, configuration Changes are made, software is installed, and patches are applied. In such an environment, the VM state is akin to a tree: At any point, execution can go into N different branches where multiple instances of a VM can exist at any point in this tree at any given time. VMs are allowed to roll back to previous states in their execution (e.g., to fix configuration errors) or rerun from the same point many times (e.g., as a means of distributing dynamic content or circulating a "live" system image).

10. KVM



Open source hypervisor based on Linux

KVM

- **Kernel module that turns Linux into a virtual Machine Monitor**
- **Merged into the Linux kernel**

QEMU
- Emulator used for I/O device virtualization

x86 virtualizations extensions
- Intel VT-x
- AMD (AMD-V)

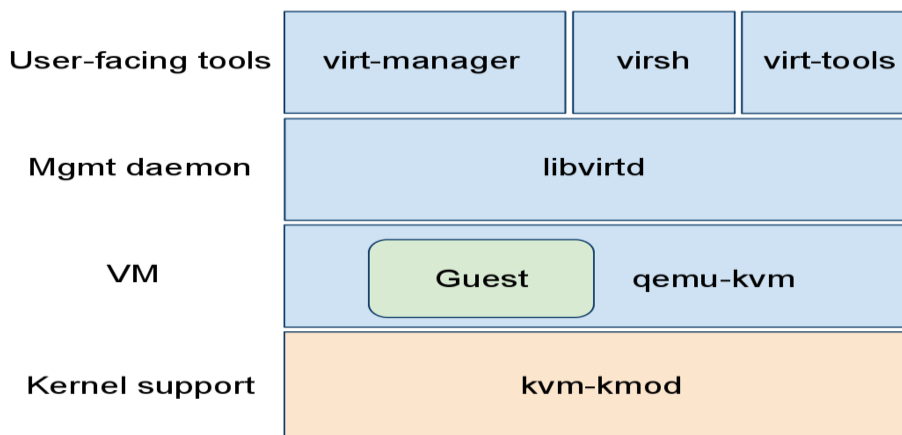
Over the past years x86 virtualization has become widespread through server consolidation and recently it is playing a role at the heart of cloud computing. KVM provides a virtualization solution with world-class performance together with the benefits of an open source platform. This post explains the key components of KVM and how they work together.

Hardware virtualization from Linux kernel

KVM is closely associated with Linux because it uses the Linux kernel as a bare metal hypervisor. A host running KVM is actually running a Linux kernel and the KVM kernel module, which was merged into Linux 2.6.20 and has since been maintained as part of the kernel. This approach takes advantage of the insight that modern hypervisors must deal with a wide range of complex hardware and resource management challenges that have already been solved in operating system kernels. Linux is a modular kernel and is therefore an ideal environment for building a hypervisor.

Full Linux hardware support for network cards, storage, and servers

Since KVM uses the Linux kernel, KVM works with network cards, storage adapters, and other hardware supported by Linux. This gives KVM excellent host hardware support that does not lag behind bare metal operating systems.



Hardware virtualization extensions provide secure and efficient way to run VM code on physical CPU

At the heart of KVM is a Linux kernel module which safely executes guest code directly on the host CPU. This is made efficient by hardware virtualization extensions, introduced in the mid-2000s by both AMD and Intel and available in almost all modern x86 processors. Virtualization extensions

added a new mode of execution that allows unmodified guests to run without giving them full access to memory and other resources.

Device emulation in user space

While guest code executes directly on the host CPU in a safe manner, most I/O accesses are trapped instead of sending them directly to host devices. The guest sees an emulated chipset and PCI bus on which both emulated and pass-through adapters can be added. KVM features paravirtualized networking, storage, and memory ballooning drivers that improve efficiency of I/O and allow adjusting the amount of RAM available to a guest at run-time.

Runs with SELinux isolation

Device emulation is performed by the qemu-kvm user space process on the host. This allows the kernel module to stay lean and focus on the most performance-critical aspects while userspace device emulation emulates hardware devices in an isolated process outside of the host kernel. The sVirt feature locks down the qemu-kvm process with SELinux Mandatory Access Control so it can only access files and resources it needs and nothing more.

Secure remote management API

Management tools need to monitor and access guests that might be running on remote hosts or locally. This is done through a set of APIs and utilities that enable applications to manipulate guests and automate management tasks. Libvirt provide the language bindings and command-line utilities for developing applications and scripting common operations.

Each host runs the libvirt daemon, which provides secure remote management APIs but it can also be configured to serve locally only and not be visible over the network. The libvirt daemon maintains guest configurations across reboot and is the central point for setting up networking and storage pools.

Systems management can be added and uses libvirt API

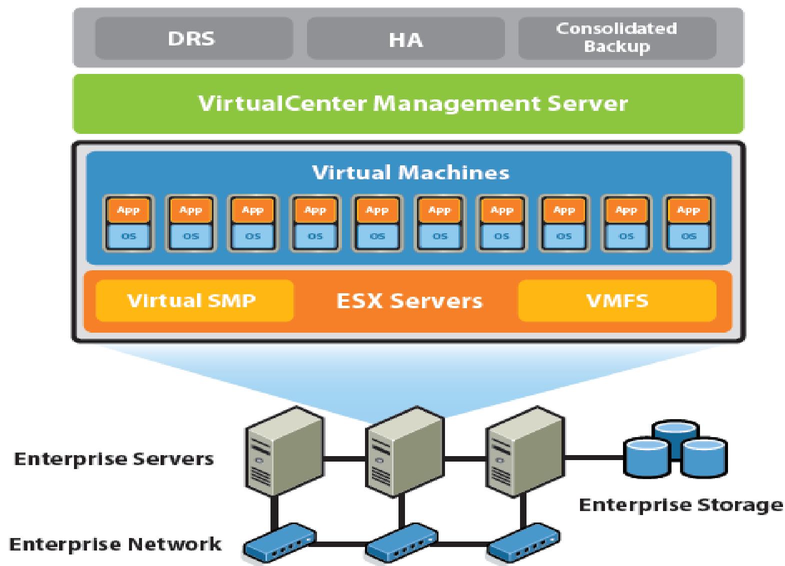
Most administration is done with tools that use the libvirt API, especially the virsh command-line tool which presents guest and host management operations. The graphical virt-manager tool can easily manage local or remote guests. Third-party management tooling such as cloud stacks can be used for higher-level datacenter or cloud management and they typically integrate with libvirt.

11.VMWare

VMware Infrastructure is the industry's first full infrastructure virtualization suite that allows enterprises and small businesses alike to transform, manage and optimize their IT systems infrastructure through virtualization. VMware Infrastructure delivers comprehensive virtualization, management, resource optimization, application availability and operational automation capabilities in an integrated offering.

VMware Infrastructure includes the following components as :
• VMware ESX Server – A production-proven virtualization layer run on physical servers that abstract processor, memory, storage and networking resources to be provisioned to multiple virtual machines
• VMware Virtual Machine File System (VMFS) – A high-performance cluster file system for virtual machines
• VMware Virtual Symmetric Multi-Processing (SMP) – Enables a single virtual machine to use multiple physical processors simultaneously
• VirtualCenter Management Server – The central point for configuring, provisioning and managing virtualized IT infrastructure
• Virtual Infrastructure Client (VI Client) – An interface that allows administrators and users to connect remotely to the VirtualCenter Management Server or individual ESX Server installations from any Windows PC

Figure 1-1: VMware Infrastructure



- Virtual Infrastructure Web Access – A Web interface for virtual machine management and remote consoles access

• VMware VMotion™ – Enables the live migration of running virtual machines from one physical server to another with zero downtime, continuous service availability and complete transaction integrity

• VMware High Availability (HA) – Provides easy-to-use, costeffective high availability for applications running in virtual machines. In the event of server failure, affected virtual machines are automatically restarted on other production servers that have spare capacity

• VMware Distributed Resource Scheduler (DRS) – Intelligently allocates and balances computing capacity dynamically across collections of hardware resources for virtual machines

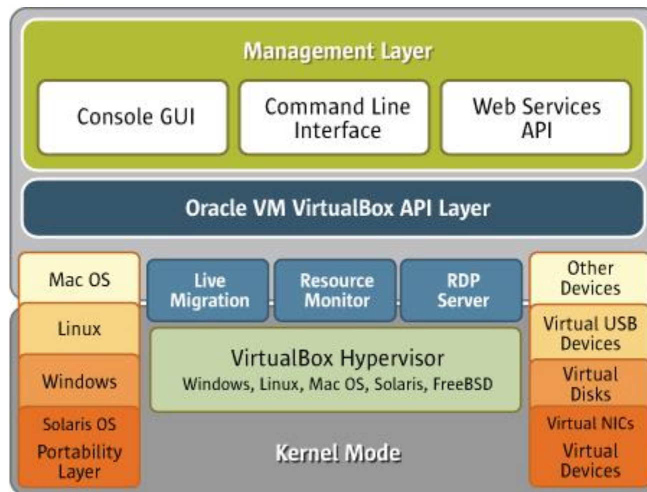
• VMware Consolidated Backup – Provides an easy to use, centralized facility for agent-free backup of virtual machines. It simplifies backup administration and reduces the load on ESX Server installations

• VMware Infrastructure SDK – Provides a standard interface for VMware and third-party solutions to access VMware Infrastructure

12. VIRTUAL BOX

Architecture

VirtualBox uses a layered architecture consisting of a set of kernel modules for running virtual machines, an API for managing the guests, and a set of user programs and services. At the core is the hypervisor, implemented as a ring 0 (privileged) kernel service. The kernel service consists of a device driver named vboxsrv, which is responsible for tasks such as allocating physical memory for the guest virtual machine, and several loadable hypervisor modules for things like saving and restoring the guest process context when a host interrupt occurs, turning control over to the guest OS to begin execution, and deciding when VT-x or AMD-V events need to be handled.



The hypervisor does not get involved with the details of the guest operating system scheduling. Instead, those tasks are handled completely by the guest during its execution. The entire guest is run as a single process on the host system and will run only when scheduled by the host. If they are present, an administrator can use host resource controls such as scheduling classes and CPU caps or reservations to give very predictable execution of the guest machine.

Additional device drivers will be present to allow the guest machine access to other host resources such as disks, network controllers, and audio and USB devices. In reality, the hypervisor actually does little work. Rather, most of the interesting work in running the guest machine is done in the guest process. Thus the host's resource controls and scheduling methods can be used to control the guest machine behavior.

In addition to the kernel modules, several processes on the host are used to support running guests. All of these processes are started automatically when needed.

- VBoxSVC is the VirtualBox service process. It keeps track of all virtual machines that are running on the host. It is started automatically when the first guest boots.
- vboxzoneacess is a daemon unique to Solaris that allows the VirtualBox device to be accessed from an Oracle Solaris Container.
- VBoxXPCOMIPCD is the XPCOM process used on non-Windows hosts for interprocess communication between guests and the management applications. On Windows hosts, the native COM services are used.
- VirtualBox is the process that actually runs the guest virtual machine when started. One of these processes exists for every guest that is running on the host. If host resource limits are desired for the guest, this process enforces those controls.

Interacting with Oracle VM VirtualBox

There are two primary methods for a user to interact with VirtualBox: a simple graphical user interface (GUI) and a very complete and detailed command-line interface (CLI). The GUI allows the user to create and manage guest virtual machines as well as set most of the common configuration options. When a guest machine is started from this user interface, a graphical console window opens on the host that allows the user to interact with the guest as if it were running on real hardware. To start the graphical interface, type the command VirtualBox at any shell prompt. On Oracle Solaris, this command is found in /usr/bin and is available to all users.

13.HYPER-V ARCHITECTURE

Hyper-V is a hypervisor-based virtualization platform and an enabling technology for one of Windows Server 2008 R2's marquee features, Live Migration. With Hyper-V version 1.0, Windows Server 2008 was capable of Quick Migration, which could move VMs between physical hosts with only a few seconds of down-time. With Live Migration, moves between physical targets happen in millisecond, which means migration operations become invisible to connected users. To find out new features and improvements in Windows Server 2008 R2 Hyper-V.

The hypervisor is the processor-specific virtualization platform that can host multiple virtual machines (VMs) that are isolated from each other but share the underlying hardware resources by virtualizing the processors, memory, and I/O devices.

Guest operating systems running in a Hyper-V virtual machine provide performance approaching the performance of an operating system running on physical hardware *if* the necessary virtual server client (VSC) drivers and services are installed on the guest operating system. Hyper-V virtual server client (VSC) code, also known as Hyper-V enlightened I/O, enables direct access to the Hyper-V "Virtual Machine Bus" and is available with the installation of Hyper-V integration services. Both Windows Server 2008 R2 and Windows 7 support Hyper-V enlightened I/O with Hyper-V integration services. Hyper-V Integration services that provide VSC drivers are also available for other client operating systems.

Hyper-V supports isolation in terms of a partition. A partition is a logical unit of isolation, supported by the hypervisor, in which operating systems execute. The Microsoft hypervisor must have at least one parent, or root, partition, running Windows Server 2008 R2. The virtualization stack runs in the parent partition and has direct access to the hardware devices. The root partition then creates the child partitions which host the guest operating systems. A root partition creates child partitions using the hypercall application programming interface (API).

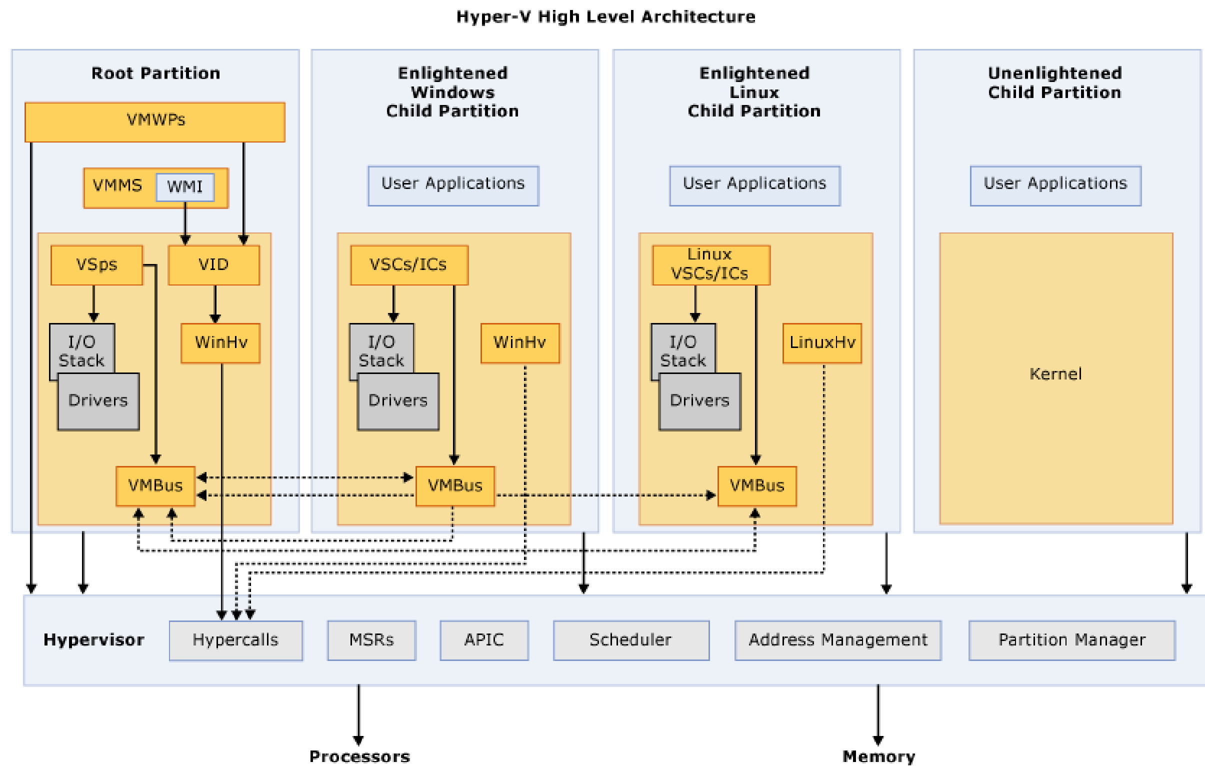
Partitions do not have access to the physical processor, nor do they handle the processor interrupts. Instead, they have a virtual view of the processor and run in a virtual memory address region that is private to each guest partition. The hypervisor handles the interrupts to the processor, and redirects them to the respective partition. Hyper-V can also hardware accelerate the address translation between various guest virtual address spaces by using an Input Output Memory Management Unit (IOMMU) which operates independent of the memory management hardware used by the CPU. An IOMMU is used to remap physical memory addresses to the addresses that are used by the child partitions.

Child partitions also do not have direct access to other hardware resources and are presented a virtual view of the resources, as virtual devices (VDevs). Requests to the virtual devices are redirected either via the VMBus or the hypervisor to the devices in the parent partition, which handles the requests. The VMBus is a logical inter-partition communication channel. The parent partition hosts Virtualization Service Providers (VSPs) which communicate over the VMBus to handle device access requests from child partitions. Child partitions host Virtualization Service Consumers (VSCs) which redirect device requests to VSPs in the parent partition via the VMBus. This entire process is transparent to the guest operating system.

Virtual Devices can also take advantage of a Windows Server Virtualization feature, named Enlightened I/O, for storage, networking, graphics, and input subsystems. Enlightened I/O is a specialized virtualization-aware implementation of high level communication protocols (such as SCSI) that utilize the VMBus directly, bypassing any device emulation layer. This makes the communication more efficient but requires an enlightened guest that is hypervisor and VMBus aware. Hyper-V enlightened I/O and a hypervisor aware kernel is provided via installation of Hyper-V integration services. Integration components, which include virtual server client (VSC) drivers, are also available for other client operating systems. Hyper-V requires a processor that includes hardware

assisted virtualization, such as is provided with Intel VT or AMD Virtualization (AMD-V) technology.

The following diagram provides a high-level overview of the architecture of a Hyper-V environment running on Windows Server 2008.



Overview of Hyper-V architecture

- APIC – Advanced Programmable Interrupt Controller. A device which allows priority levels to be assigned to its interrupt outputs.
- Child Partition – Partition that hosts a guest operating system. All access to physical memory and devices by a child partition is provided via the Virtual Machine Bus (VMBus) or the hypervisor.
- Hypercall – Interface for communication with the hypervisor. The hypercall interface accommodates access to the optimizations provided by the hypervisor.
- Hypervisor – A layer of software that sits between the hardware and one or more operating systems. Its primary job is to provide isolated execution environments called partitions. The hypervisor controls and arbitrates access to the underlying hardware.
- IC – Integration component. Component that allows child partitions to communication with other partitions and the hypervisor.
- I/O stack – Input/output stack.
- MSR – Memory Service Routine.
- Root Partition – Manages machine-level functions such as device drivers, power management, and device hot addition/removal. The root (or parent) partition is the only partition that has direct access to physical memory and devices.
- VID – Virtualization Infrastructure Driver. Provides partition management services, virtual processor management services, and memory management services for partitions.
- VMBus – Channel-based communication mechanism used for inter-partition communication and device enumeration on systems with multiple active virtualized partitions. The VMBus is installed with Hyper-V Integration Services.

- VMMS – Virtual Machine Management Service. Responsible for managing the state of all virtual machines in child partitions.
- VMWP – Virtual Machine Worker Process. A user mode component of the virtualization stack. The worker process provides virtual machine management services from the Windows Server 2008 R2 instance in the parent partition to the guest operating systems in the child partitions. The Virtual Machine Management Service spawns a separate worker process for each running virtual machine.
- VSC – Virtualization Service Client. A synthetic device instance that resides in a child partition. VSCs utilize hardware resources that are provided by Virtualization Service Providers (VSPs) in the parent partition. They communicate with the corresponding VSPs in the parent partition over the VMBus to satisfy a child partitions device I/O requests.
- VSP – Virtualization Service Provider. Resides in the root partition and provide synthetic device support to child partitions over the Virtual Machine Bus (VMBus).
- WinHv – Windows Hypervisor Interface Library. WinHv is essentially a bridge between a partitioned operating system's drivers and the hypervisor which allows drivers to call the hypervisor using standard Windows calling conventions
- WMI – The Virtual Machine Management Service exposes a set of Windows Management Instrumentation (WMI)-based APIs for managing and controlling virtual machines.