Virtual Machines

Glenn Story

Last Updated November 11, 2019

Introduction

This is about virtual machines. I have been working with virtual machines since the days of CP-67. I find them endlessly fascinating.

A thorough understanding of them requires a deep understanding of both the hardware that they emulate and the operating-systems with which they must interact. I have nevertheless endeavored to write this at a level that can be understood by someone who has a working experience with computers, but who is not necessarily a computer scientist or engineer.

I should say at the outset that I am a former employee of VMware, a well-known and successful supplier of virtual-machine products. I will be mentioning VMware implementation techniques frequently. However I have intended to rely solely on public information, such as VMware's public documentation. See the References section at the end of this document for specifics.

Definition of "Virtual Machine"

What exactly is a "virtual machine?" To paraphrase a Presidential quote, that depends on how you define those words, like "virtual" and "machine."

The word "virtual" (and its nominal form, "virtualization") have become buzz words lately, and as such are largely devoid of precise meaning. Nonetheless, I would say that when one is properly using the term "virtual," one is referring to the software simulation of a hardware system or component.

The most common use of "virtual" is "virtual memory." Almost all modern operating systems support virtual memory. Each running program in such an operating system has the illusion that it has exclusive use of a private block of memory that starts at address zero and generally extends for 4 GB (gigabytes). Of course, at the hardware level, most desktop computers have less than 4 GB of *real* memory to share between *all* the programs running in the computer. The illusion is carried out by software within the operating-system kernel. This illusion--or simulation--requires real hardware, including real memory. That demonstrates a general point: even though virtualization is carried out by software, it requires hardware support. But of course, all software requires hardware.

Other examples of the use of "virtual" are "virtual private network (VPN)" and "virtual circuit," both networking terms.

We also hear of "storage virtualization," a term used by storage-software vendors such as VERITAS.

The word "machine" is a kind of computer slang for "computer" or "computer system." We often say "that is my machine" or "that is a lab machine," etc.

Thus, a "virtual machine" is a software simulation of a computer system.

Uses of Virtual Machines

Multiple OS's per Computer

One of the fundamental uses of a virtual machine is to run multiple operating systems on the same computer. As an example, the computer I am writing this document on runs Linux on the "real" computer, but then I am running two other versions of Linux plus one version of Microsoft Windows, each in its own virtual machine.

Time Sharing and Concurrent Execution of Multiple Operating Systems

Let me give some examples from my own personal experience of uses of virtual machines. (A more detailed memoir of my working career can be found at http://glennastory.net/cm/.)

My first exposure to virtual-machine technology was a commercial variant of CP-67 when I worked for Bank of America. (Specific products, such as CP-67 will be described in more detail in the following section on the history of virtual machines.) The bank used this technology simply as an implementation of an interactive time-sharing system, i.e. a mechanism to allow multiple users to have simultaneous access to a large mainframe computer. Each user had his or her own terminal and could run his or her own programs from it. Each user also had his or her own virtual machine, but that fact was not obvious to the user. Nor was the presence of a virtual machine a necessary ingredient for the time-sharing application that was being provided. We used this environment to develop software to be run in the bank's mainframe environment. The interactive nature of this environment made it much more productive than the traditional software-development environments in general use at the time. This creation of an interactive environment continued to be an important use of virtualization up until the time that personal computers became prevalent and powerful enough that they could provide an interactive environment without resorting to virtualization or any other technique for time-sharing.

Later in my career I worked for John Breuner Company. We started using VM/370 when we decided to convert from one operating system (DOS/VSE) to another (OS/VS1). We simultaneously ran each of these operating systems in virtual machines, to allow for the conversion. We also took advantage of the presence of VM/370 to use its interactive characteristics for software development.

When I left Breuners I returned to Bank of America. This time I worked on a project that made financial data available to small businesses interactively. Each business had its own terminal access to the system. Although they didn't realize it, each of them had their own virtual machine, running on VM/370.

Subsequently I worked for ITEL, (not to be confused with Intel). ITEL was a company that sold mainframe computers that competed with IBM's. I was doing operating-system programming at that time and we used VM/370 so that each of us could have our own virtual machine running the operating system we were working on.

From these examples we can see several common uses of virtual machines:

1. Providing an interactive environment for software development, or even application execution.

2. Providing a vehicle to run multiple production operating systems at the same time on the same computer.

3. Providing a vehicle to allow the testing and debugging of operating-system code without dedicating a real machine to each test.

Server Consolidation

All of the above examples were done using IBM's VM/370 or its predecessors. When VMware introduced virtual machines to the Windows desktop environment some twenty years later, the uses they envisioned were largely the same. However, it was not long before a new use came to light: the ability to reduce the number of separate server machines into virtual machines running on a single real computer.

This concept, called "server consolidation" deserves further discussion, because it has come to be the most significant use of virtualization in recent years. It is this server consolidation that has led to the excitement over virtualization in large companies' IT departments and it is also what has led to the success of VMware and its emulation by a number of more recent companies.

One of the key features of any modern operating system is the ability to run multiple programs at the same time. This means that I should, in theory, be able to run multiple applications on the same machine. Most large IT departments run web servers, email servers, application servers, and database servers. They should all be able to run on the same computer, assuming only that the computer has the capacity for them all.

In practice this is seldom done. There are several reasons for this: First, often these applications require different operating systems. The mail server may be running Microsoft Exchange, which, of course, runs on Windows. The database server might be DB2, which would run on an IBM system. File servers and application servers might be on Linux, or Solaris. Even if two applications run on the same operating system, they often require different tuning and configuration. And even if that can be solved, it is often safer to run each application on a different machine, simply because one application can crash in a way that brings the whole system, including any other applications, down.

So, as a consequence, most IT shops run one application per server. If these servers are busy all the time it is a good use of resources, but often a given application is only busy intermittently and separating each application to a separate machine, reduces resource utilization, increasing not only equipment costs but energy and cooling costs as well.

Server consolidation allows a user to continue to run each application in a separate machine, but now they are virtual machines so several of them can be run on the same real computer. Thus the total amount of hardware can be significantly reduced.

Another factor is the fact that each physical machine can have a slightly different hardware configuration: a different amount of memory, a different brand of network card, etc. This means that if you wish to move an application to a different machine, you have to do substantial reconfiguration and testing.

Each virtual machine, on the other hand, has its own set of virtual hardware. Moreover this virtual configuration is stored as a series of parameters in one or more disk files on the host computer. It is therefore simple to move a vritual machine to another host computer without changing its virtual configuration. Thus, from the virtual-machine's point of view the (virtual) hardware hasn't changed as a result of the move.

These benefits have proven themselves to be real and of significant value, so that almost all large IT organizations today are at least experimenting with the use of virtualization.

Cloud Computing [14]

Someone recently remarked that "cloud computing" is the buzz word for 2009 just as "virtualization" was the buzz word for 2008. "Cloud computing" refers to the idea that resources (such as processors) can be dynamically accessed across a network (such as the Internet) as needed instead of being dedicated to an application. The term "cloud" presumably derives from the fact that networks are drawn as a cartoon-like cloud. We often hear of acquiring resources "from the cloud" when speaking of cloud computing.

There are three distinct types of cloud computing:

1. Software as a Service: I am writing this document using Google Docs, which is an example of software as a service. Whereas traditionally I would have used some word-processing program running on my local computer, now I am typing these words into a web browser and the actual word processing (as well as document storage) is being done on some Google server. This is a typical example of software as a service.

2. Platform as a Service: Several companies provide services on which applications can be written using interfaces specific to the cloud-computing environment. Google's "Google Application Engine" (GAE) is a major example of Platform as a Service.

3. Infrastructure as a Service: In this case a provider supplies a virtual machine upon which the customer runs a guest OS and application. Amazon's "Elastic Computing Cloud" (EC2) is a major example of Infrastructure as a Service.

Whereas any of these three types can be and are implemented on virtual machines (since *any* application can be run on a virtual machine) , the third type (Infrastructure as a Service) is tied specifically to virtualization. If my application needs additional processors during peak periods, then I simply "rent" a virtual machine from some provider across the Internet.

A note on the cloud and the consumer: When I originally wrote this, the “cloud” buzzword was largely used with respect to enterprise computing, not consumer products and services. That’s changed since then; “cloud” has become a widely used term with respect to consumer offerings. But mainly it refers to *cloud storage* which means storing your files, music, pictures, etc., “in the cloud” i.e. on some system accessed via the Internet. There is some consumer cloud computing (such as Google Docs mentioned above. (But even “Google Docs” has been rechristened “Google Drive” which puts more of an emphasis on cloud storage than on cloud computing.) Anyway we are more interested in cloud *computing* than cloud *storage* in this paper.

Cross-Platform Execution

The second fundamental use of virtualization is to run software from multiple architectures on one machine. Generally, programs written for one hardware platform or operating system will not run on any other. (I refer here to "object code" not "source code" for those of you who understand this distinction.) Let me give two examples of how virtualization can alleviate this problem.

Platform Sharing

The Apple Macintosh is a popular computer system with many, due to its elegant design, and ease of use. But, because it represents only a small fraction of the overall desktop and laptop market, there are many more applications written for Microsoft Windows, the market leader. One gives up all these Windows-only applications if one chooses a Macintosh. But with the addition of virtualization, it becomes possible to run a Windows virtual machine on a Macintosh computer. Now you can run the applications from both operating systems at the same time on the same system.

Programming-Language Environments

As another example, some programming languages, most notably Java, have been designed to create programs only for one type of machine--a fictitious machine implemented only as a virtual machine. The programs written for this virtual machine (for example the Java Virtual Machine) can then be run on any real computer that has the software to create such a virtual machine. Whether this class of solution should truly be included in a discussion of virtual machines is arguable. Originally I considered the use of the term "virtual machine" in the name "Java Virtual Machine" to be a misuse of the term. In fact, Microsoft's similar implementation in .NET is classified by them as a "framework" rather than a "virtual machine". I have decided that, based on the definitions I used, and also based on the implementation techniques used (such as simulation and binary translation--both described later) that solutions such as the Java Virtual Machine have a rightful place in a discussion of virtual-machine technology.

Goals of Virtualization

VMware [7] has listed four specific goals of virtualization:

1. **Compatibility:** The virtual machine must perform in an identical fashion to the real machine it is emulating. In the case of fictional target machines, such as the Java Virtual Machine, then at least each virtual machine should be compatible with the others of the same type. The one factor that cannot be made identical is timing. A software-based virtual machine is always going to run more slowly than a hardware-based real machine of the

same type.

2. **Isolation:** Each virtual machine should be completely isolated from all other virtual machines and from the real machine on which it is running. A faulty or malicious application on one virtual machine should not damage any other virtual machines. This feature is required for correct operation of any virtual machine but is particularly important when virtualization is used for server consolidation.

3. **Encapsulation:** Each virtual machine should be encapsulated so that it exists as a separate entity. This is primarily done by defining each virtual machine using a set of disk files that completely describe that virtual machine. This makes it simple to move that virtual machine to another real machine.

4. **Hardware independence:** The virtual I/O devices, memory, etc. that make up a virtual machine are generally separate from and independent of the I/O devices on the real machine. As an example I may have SCSI disk drives in a virtual machine, but IDE disks on the host computer.

History

Now let us turn to an examination of how the idea of virtual machines came about and developed.

CP-40

In its day, the IBM System/360 was quite an innovative product line. It incorporated a large number of ideas that were either completely novel, or had only been used in research computers in the past.

One of the most important ideas incorporated into the System/360 was the idea that there would be a large range of machines, from small to large, slow to fast, cheap (relatively) to expensive that all ran the same machine instructions--that all have the same "architecture," to use a term that IBM coined to describe this aspect of System/360 design. Prior to the advent of the System/360 a computer customer would be required to completely rewrite his application programs if he moved from a smaller machine (such as the IBM 1401) to a larger machine (such as the IBM 7094). Now, one could move from a smaller System/360 model (such as the Model 30) to a larger model (such as the Model 50) and the old programs would continue to run--only they would run faster.

In 1964, the IBM Scientific Laboratory in Cambridge Massachusetts took a System/360, Model 40, and modified it to include an additional feature: Dynamic Address Translation (DAT) This was a hardware mechanism designed so that the memory addresses referenced in a program were *translated* by DAT to *different* addresses for presentation to the memory hardware. This was the basis for what would become known as virtual memory.

Not only did the IBM engineers create virtual memory, they created a virtual machine. This software could create multiple simulated "machines" each with its own (virtual) processor, memory, and I/O devices, and each one capable of running its own operating system. (At that time there were two main IBM operating systems: OS and DOS with several varients. In fact IBM invented the term "operating system" for use with the System/360.)

They called the softwre system that created these VMs, "CP-40." It had two main components:

CP, standing for Control Program. "Control Program" was the general term IBM used at that time for what we would today call the "kernel." It is the fundamental piece of system software used to control and manage a modern computer system, providing easier access to I/O devices and the ability to run multiple programs at the same time. It was CP that in essence created the virtual machines. It provided the software layer that simulated the various components making up the virtual machine.

CMS, the Cambridge Monitor System. Recall that CP-40 was created at the IBM Cambridge Scientific Laboratory. "Monitor" is a term that was then used for system software--a kind of primordial operating system. I would distinguish a monitor system from a true operating system in that monitor systems didn't support multi-programming, the running of more than one program concurrently. (By my definition, MS-DOS would be a monitor system rather than an operating system.)

There was no idea of a "personal computer" in those days. For an individual to have personal control of an IBM mainframe computer (and mainframes were all that existed then) would have normally been prohibitively expensive. But a virtual machine is--virtually--free. After all it is just software. You do need some terminal device to act as the virtual console, but a terminal device cost hundreds of dollars, versus millions of dollars for a mainframe computer system.

But now that you have a virtual machine, what are you going to run? You need an operating system, and all of the operating systems of that day were "batch" operating systems--operated by feeding in decks of punched cards, not run interactively as modern systems are. (And yes, CP-40 did support virtual card readers and virtual card punches.)

It would be much nicer to have a computer system that was interactive, so that you could type in commands and get responses. And that's exactly what CMS was. As I have alluded, it only ran one program at a time, but you could interact with it.

CP-67

The System/360 Model 40 that the engineers modified at the Cambridge Scientific Labs was a one-of-a-kind modification. But it was sufficiently successful that IBM subsequently came out with a new production model: the Model 67 that was identical to the previously-available Model 65 with the addition of DAT.

The Cambridge engineers then produced CP-67, a virtualization system that was conceptually identical to CP-40, including the CP and CMS components.

VM/370

Just as System/360 was a creation of the 1960s, so the System/370 was a creation of the 1970s. It was the successor line of computers to the 360.

The System/370 incorporated a number of incremental improvements to the 360, but the two systems were largely compatible. Certainly almost any System/360 program would run unchanged on a 370.

One major change from the 360 to the 370 was the spread of DAT and virtual memory (called at that time "virtual storage.")

Operating system software had to be upgraded to support virtual storage, so OS became OS/VS and DOS became DOS/VS.

And, more important for our story, CP-67 was recreated from a research system, to a commercial system, and christened "VM/370".

VM/370 had the same components as CP-67: CP, the control program, whose purpose was unchanged, and CMS. CMS was re-designated as "*Conversational* Monitor System" but its purpose, too, was unchanged.

There was also introduced a second monitor-like system, designed to be run in a virtual machine: RSCS (Remote Spooling Communication Subsystem). I won't go into the details, but suffice it to say RSCS was used for network communications between VM/370 and certain types of terminals, and eventually between two running copies of VM/370 on two separate (real) computers.

It is important to note that VM/370, and in particular CP, ran directly on the "bare metal", that is to say it ran directly on the hardware and had to do all of the things an operating-system kernel must do: control memory and I/O, process interrupts, etc. This is in contrast to most of the other systems we shall discuss, where the CP-equivalent component runs under the control of some other operating system.

IBM made an attempt in the mid 1970s to discontinue VM/370 as one operating system too many. There was an outraged protest from customers and IBM ultimately kept it alive. It is still available today, on IBM's current line of mainframes, the Z-series. VM/370 is now called z/VM [12].

Amdahl Hypervisor

Gene Amdahl was one of the architects of the IBM System/360. Later he founded his own company that made "IBM Clones" -- computers that were architecturally the same as IBM mainframes.

One of the pieces of software to come out of the Amdahl Corporation was called the Hypervisor. Essentially it was an extension of VM/370. The details are not important; the techniques used in that product were never used again.

I mention this product only because of its name, Hypervisor. To my knowledge that is the first time that term was ever used. It has since become a standard term in virtuallization literature. It has come to mean the component that actually simulates hardware that is being virtualized.

UCSD P-System

Pascal was a programming language developed by Niklaus Wirth, in 1970. Originally Pascal programs were, as in most programming languages then and now, translated into machine language by a piece of software called a compiler. This compiled program would only run on the type of computer for which it was compiled. A Pascal program compiled on a System/360 would not run on, say, a Burroughs or CDC computer. One would have to take the original ("source") program to the new machine and compile it again.

Then, in 1977 [2] researchers at the University of California at San Diego came up with a scheme to translate the source code, not into a "real" machine language, but into a made-up machine language called P-code. How was this P-code, then, to be executed? The answer is through an interpreter that simulated the made-up machine that could execute P-code directly. In short, it was a virtual machine, although that term was never used. Thus, one could compile a Pascal program to P-code on one type of computer and then run it on another type.

The P-system, like the other systems we have discussed so far, ran on the "bare metal", i.e. not under some other operating system.

As a language, Pascal was limited in functionality. (It was largely designed as a teaching language.) Likewise the P-system, which doubled as an operating system and virtual machine system, was also limited. Both have faded from the scene.

Java

Java is another programming language. Unlike Pascal, for which the P-system virtual machine implementation was only one of several implementations of the Pascal language, Java was designed, starting in 1991[3] from the beginning to run in a virtual machine. It's even *called* a virtual machine: the "Java Virtual Machine".

As with the P-system, the Java compiler translates Java source code into a made-up machine language, this time called "bytecode." Then the Java Virtual Machine (or "JVM") executes the bytecode to run the program.

Java was designed by Sun Microsystems. They supplied compilers and JVMs, but other vendors (notably IBM) supply competing versions of this software. That was fine with Sun, as long as the other versions remain 100% true to the original design.

The advantage, as with the UCSD P-system, is that a Java program compiled on one type of computer will work on a different type of computer, provided only that a JVM has been written for that other computer. Sun called this "Compile once; run everywhere."

Java JVMs (at least those written to date) are designed to run under a host operating system that supplies memory management, I/O support etc. For example, a file read or written by a Java application is read from or written to the operating system's file system.

Microsoft .NET CLR

Microsoft's .NET is a software environment that was originally announced in 2000. One of .NET's key components is the CLR or Common Language Runtime. Although Microsoft does not use the term, "virtual machine", the CLR is, for all intents and purposes a hypervisor.

.NET is similar in many ways to Java. Like Java (and the P-system before it) .NET was designed to act as a layer between the softare and the underlying hardware so that programs could be moved from one platform to another. A significant difference between .NET and Java is that .NET supports multiple programming languages, whereas Java supports only one language--the Java programming language. (Thus, "Java" can refer to either the programming language or the runtime environment (the Java Virtual Machine), whereas ".NET" refers only to the runtime environment.)

VMware

If you look back to the preceding systems you will note that all the systems *after* VM/370 have to do with programming languages. Until recently, VM/370 was the one and only general-purpose virtual-machine software system, creating virtual machines that are virtualizing real hardware architectures. Why has nothing come along since then?

The answer (which will be discussed more fully in subsequent sections) has to do with the underlying machine language or architecture of the IBM System/370 for which VM/370 was written. That architecture makes it easy to write virtualization software, again for reasons to be discussed later. Today the predominant architecture is the so-called Intel x86 architecture, which began with the 8086 (and 8088) which formed the CPU of the first IBM personal computer. The x86 has some important deficiencies when it comes to creating virtualization software. For that reason, and because of the general processing speeds of early x86 processors, virtualization was much more difficult. Moore's law (that processing speeds double every 18 months) has taken care of the performance problem.

The architectural deficiencies of the x86 processors were overcome by adopting new virtualization softare techniques. The work was done by Mendel Rosenblum[4], an associate professor of computer science at Stanford University and his students and fellow researchers. That work led to the creation of the VMware company in 1998.

VMware started with a single virtual-machine product and has expanded to a number of related products for running and managing virtual machines.

VMware has had unparalleled success with its products and has created a widespread acceptance of virtualization as a tool for managing the large collection of servers that most large businesses own today. Because of that success, a number of other companies, including Microsoft and Sun Microsystems, have started producing virtualization products.

Apple Macintosh

You may be surprised to see a hardware product mentioned here. But the CPU architecture upon which the Macintosh is built has changed twice over the life of the product line: The original Macintosh was based on the Motorola 68000 processor. Later models were based on the Power PC, developed co-operatively by Apple, IBM, and Motorola. Now, Macintoshes are based on the Intel x86 family of processors.

At the time of each conversion, Apple wanted its customers to be able to continue running their old software (written for the old architecture) on the newer machines. At each conversion, they solved that problem using the CPU virtualization techniques to be described here.

Other companies, notably Hewlett Packard, and Tandem Computers, have faced similar problems and solved them in similar ways.

Xen

The open source movement has been an intriguing and important aspect of the development of software for the past twenty or more years. It was inevitable that an open-source virtualization product would emerge. In fact it was probably inevitable that *several* open-source products would emerge. Arguably the most important of these is Xen (pronounced, "zen") which was first released in 2003. [5]

Microsoft Virtualization Products

Microsoft originally had two products, *Virtual PC* and *Virtual Server,* which create client and server virtual machines respectively The latter has recently been replaced by a new product, *Hyper-V*.

Virtual PC and Virtual Server were acquired by Microsoft when they bought the startup company, Connectix, in 2002. [11] Hyper-V was developed in-house by Microsoft, although they are not saying what, if any, technology, they brought over from the Connectix products. All these Microsoft products support both Microsoft and non-Microsoft guest OS's. The host must be a version of Microsoft Windows.

Oracle VirtualBox

VirtualBox began as a commercial product by a company called Innotek. They later released VirtualBox as free open source software.

Innotek was eventually acquired by Sun Microsystems. Sun, in turn, was acquired by Oracle, which continues to support the product as a free open-source product.

KVM and QEMU

QEMU is a free open-source infrastructure for supporting virtualization. It can be used to build solutions based on emulation (described below) or it can interface with KVM, another free open-source package that runs in the real machine’s kernel. For both KVM and QEMU the host operating system must be Linux.

Types of Virtualization

From the above historical sampling we can distinguish two types of virtualization:

1. OS-related: These products create a simulation of a real computer so that an operating system can be run within that virtual machine. These products include VM/370, VMware, VirtualBox, and Xen, among others.

2. Language-related: These products are designed to create an environment in which a programming language can run. These products include UCSD Pascal, Java, C#, Visual Basic.net et al..

Terminology

The following terms will be used throughout this document.

Bare Metal: A slang term meaning a computer environment that does not require an operating system. Some virtual-machine products run on the bare metal, whereas others run under a host operating system.

Guest: The operating system running in a virtual machine, or sometimes the virtual machine itself. "Guest OS". Contrast with "host" q.v.

Host: The operating system running on the real machine, or sometimes the real machine itself. "Host OS". Contrast with "guest" q.v.

Hypervisor: The software component that creates the virtual machine. When a guest "thinks" it is interacting with the hardware, it is frequently actually interacting with the hypervisor. Also called the "virtual-machine monitor" or simply "monitor."

Monitor: See "Virtual-Machine Monitor".

Real Machine: The actual computer system upon which a hypervisor runs in order to create virtual machines.

Virtual Machine: The software simulation of a computer system.

Virtual-Machine Monitor: The software component that creates virtual machines. Also known as a "hypervisor".

Three additional notes on terminology:

1. The terms "emulate" and "simulate" and their related nominal forms, have a specific meaning in computer terminology. However such terms are largely historical in use. I use those terms in this document in their standard English-language definitions.

2. Many computers incorporate microcode (a level of software *below* (and used to create)

machine language). The distinction between hardware and microcode is not necessary in this document so whenever I use the term "hardware" in this document, it may be taken to mean "hardware *or microcode*." (Microcode will be specifically--if briefly--mentioned in relation to the "Virtual Machine Assist" microcode associated with VM/370.)

3. The terms "virtual-machine monitor" and "hypervisor" mean essentially the same thing; I use the terms interchangeably in this paper, although I usually prefer the term "hypervisor".

How Virtual Machines are Created

A virtual machine, like a real computer, has basically three types of hardware components:

- One or more CPUs (Central Processing Units)
- Memory
- I/O (Input/Output) or peripheral devices

Each of these must be somehow emulated in a virtual machine. The subsequent sections of this paper will discuss each of these in turn.

The Virtual CPU

The job of the CPU is to execute programs. Programs are made up of instructions that are encoded in a numeric form and stored in memory. The CPU fetches these instructions from memory, decodes them to determine what operation is to be performed and then executes that operation. Once that is completed, the CPU begins the cycle over again by fetching the next instruction from memory. This is known as the fetch-decode-execute cycle. (Modern CPUs typically fetch, decode, and execute multiple instructions in parallel to achieve higher performance, but conceptually they still run this cycle one instruction at a time.)

The set of instructions a CPU can decode and execute, the way in which each instruction is encoded and the exact effect it has are referred to variously as the CPU's architecture or machine language. (Technically "architecture' is a broader term that encompasses both the machine-language instruction set as well as other aspects of the CPU's behavior.)

There are "families" of processors. Each processor in a family shares a common architecture and machine language. Processors from different families typically have *different* machine languages and architectures.

The first family of processors sharing a common architecture was the IBM System/360. There were several models of System/360, the Model 30, Model 40, etc. They each had different performance and cost characteristics but they all shared the same architecture and machine language. Thus a program written for a System/360 Model 30, could be moved to, say, a Model 65, and it would still run, albeit significantly faster. Prior to that time, each computer had its own machine language and if you outgrew your computer and had to buy a bigger one, you had to re-write all your software in the new machine language.

In 1978, Intel announced the 8086 processor. [6] Subsequently they released the 80286, 80386, 80486 and 80586 (which they rechristened the "Pentium"). This family of processors has come to predominate the entire computer industry today.

When we are running a virtual machine, there are actually *two* architectures to consider: (1) the architecture of the real machine (on which the virtualization software is run) and (2) the architecture of the virtual machine. This creates two situations to be handled when implementing a virtual CPU: (1) In one case the architecture of the real and virtual machines are the same. (2) In the other case the real machine and virtual machine have *different* architectures.

In the second case the software that is creating the virtual machine (often referred to as the *hypervisor* or the *virtual-machine monitor*) must translate every single instruction from the virtual machine's machine language to the real machine's machine language. This translation can cause significant overhead and therefore clever strategies have been invented to optimize this translation.

When the Real and Virtual Architectures Differ

Let us first, therefore examine the techniques used when the real and virtual machine languages differ.

Interpretation

Just as the hardware in the CPU executes the "fetch-decode-execute" cycle described previously, it is possible to write a piece of software that will fetch an instruction from memory, decode its meaning and "execute" that instruction by executing a series of instructions written for the real computer. This technique is known as "interpretation."

Interpretation is simple in principle but complex in practice because the machine language being interpreted often contains hundreds of instructions as well as multiple addressing modes and other variables. Moreover, if one virtual instruction is interpreted by executing multiple real instructions, the resulting performance is bound to be significantly slower than running the real instructions. This is compounded by the fact that the fetch and decode steps must also be done using multiple instructions for each virtual instruction.

Despite these difficulties, several of the virtualization systems described earlier used interpretation.

Binary Translation

Interpretation essentially translates a virtual instruction into one or more real instructions *each time the virtual instruction is executed.* Programs are made up of "loops": sequences of instructions that are executed over and over again. Wouldn't it be more efficient to do the translation of the virtual instructions--especially those in loops--*once* and use the translated instruction over and over, each time the instruction is executed? This technique is known as "binary translation."

There are two variations of binary translation.

In one variation, the entire virtual program is translated in one step. This step is done separately from, and prior to the execution of the program. The translated program is then generally stored on disk along with the original program.

In the other variation, small chunks of the program are translated only when they are needed. This technique is referred to as "just in time" or "JIT" translation. In this technique the translated code is usually stored in memory and discarded when the program execution is finished.

When Real and Virtual Architectures are the Same

You may think that the problem in the case where virtual and real architectures are equal is trivial or even non-existent. After all, why can't the hypervisor simply *execute* the instructions in the virtual machine directly on the real machine's CPU? For the most part it can. The problem occurs when the virtual machine's operating system tries to execute what is known as a *privileged instruction.*

You see, every computer architecture that supports full-functioned operating systems has certain instructions that are reserved for only the operating system *kernel* (the key part of the operating system that has direct control of the hardware). These privileged instructions have to do with managing memory and I/O devices, as well as certain aspects of the state of the CPU itself. If every application program running within the OS could access these hardware resources, then they could accidentally (or maliciously) interfere with the execution of other applications. Thus the OS kernel runs in a special mode (referred to as "kernel mode", "privileged mode" or just "priv mode") that gives the kernel full access to the resources of the system. Ordinary application programs, on the other hand, run in a different mode (referred to as "user mode" or "non-priv mode") that disallows the accessing of certain CPU features. For example a user-mode program is generally not allowed to do input/output operations, since the use of an I/O device must be controlled by a central authority--the operating system. (The kernel provides to user-mode programs ceertain *system services* by which the application can ask the kernel to perform services like doing I/O in behalf of the appllication program.)

It is one of the design principles of modern operating systems that one program should be isolated from the mistakes or deliberate intrusions of all other programs in the system.

By the same token, it is one of the design principles of virtual machine systems that one virtual machine should be isolated from the mistakes or intrusions of all other virtual machines in the system. (Refer back to the goal of "isolation" under "Goals of Virtualization, earlier in this paper.)

It follows, therefore, that the kernel of an operating system running in a virtual machine cannot be allowed to run in privileged mode or it would violate the principle of isolation: a piece of software running in privileged mode could accidentally or maliciously alter the memory or state of other virtual machines or other code running in the real machine. Yet, the kernel running in the virtual machine contains privileged instructions. How are they to be handled? By translating them. Just as *all* instructions must be translated when the virtual and real machine languages differ, so *privileged* instructions must be translated when the virtual and real machine languages are the same.

The component that does this translation is the hypervisor, and *that* component often *is* run in privileged mode.

Interpretation

Theoretically, one could solve this problem by simply interpreting *every* instruction, the way we do when the virtual machine language is different than the real machine language. But the performance penalty here is much less justified than for that of differing machine languages. That is because most instructions (even in the kernel) are non-privileged instructions and by using interpretation we lose the ability to run those non-priv instructions at "full speed" by executing them directly on the real CPU. For that reason, this technique is never used on any real implementation of a virtual-machine facility in the case where the real and virtual architectures are the same.

Trap and Emulate

On many machine architectures (including the IBM System/360 where the first virtual machine facility was run) the execution of a privileged instruction by a non-privileged component causes the computer to switch to a different piece of code (usually in the kernel) that handles the invalid situation--generally by terminating the offending program. This switch is called an "interrupt", "exception" or "trap".

In the case of virtualization, the hypervisor sets things up so that *it* gets control when such a privileged-instruction trap occurs. (This setup is itself a privileged operation, which is one reason the hypervisor generally runs in privileged mode).

Thus when a virtual-machine's kernel issues a privileged instruction, a trap occurs, the hypervisor gets control, emulates the privileged instruction and then returns control to the next instruction in the virtual machine's kernel. This technique is therefore called "trap and emulate".

VM/370 is an example of a system that used trap and emulate.

Binary Translation

"Trap and emulate" is an efficient way of catching privileged operations. But it only works if the CPU is 100% correct in terms of trapping privileged operations. The IBM System/360 and 370 were correct in this regard. The Intel x86 family (upon which most modern computers are built) is not. There are cases of an instruction where a privileged operation is trapped. But there are other cases where the operation is simply modified depending on whether the priv or non-priv state is in effect. And there are cases where the software can read some piece of privileged information but cannot write it without trapping. This would be harmless in a normal kernel, but in a virtualization environment, there may be separate virtual state that should be returned.

For these reasons, trap and emulate does not work on x86 processors. So an alternate technique is to use binary translation. It was Mendel Rosenblum and his associates at Stanford University who originated the concept of using binary translation to solve the problem of virtualizing the Intel x86 processors.

Binary translation works like this: The sequence of instructions to be translated from is read in their original locations in memory. A new block of memory is set aside for the translated instructions. In the case of non-priv instructions the "translation" is usually one-for-one: the virtual instruction is simply copied to the corresponding real-instruction location. For privileged instructions, however, the virtual instruction is translated into a sequence of instructions that will emulate the privileged operations typically by calling the hypervisor.

VMware is an example of a system that uses binary translation. (Mendel Ropsenblum, the principal inventor of binary translation for virtualizing x-86 processors became a primary founder of VMWare.)

Patching in Place

Patching in place is a variation on binary translation. In traditional binary translation, *all* instructions are re-created in a separate piece of memory. Non-privileged instructions are copied verbatim, whereas privileged instructions are replaced with different code that frequently transfers to a subroutine somewhere else in memory to emulate the privileged instructions.

In the technique that I call patching in place, non-privileged instructions are left alone and privileged instructions are replaced in their original locations with alternatives (or jumps to alternatives) as in binary translation.

VirtualBox uses this technique.

Paravirtualization

The virtualization techniques mentioned so far assume that the kernel running in the virtual machine was written to run in a real machine and therefore issues privileged instructions.

However, it is possible to produce a *modified* kernel that is designed to run in a virtual machine. Instead of issuing privileged instructions, the kernel directly interacts with the hypervisor. (By this I don't mean the automated translation described for binary translation or patch in place. In the case of paravirtualization, this is a manual process requiring human programmers to modify existing code.)

The advantage of paravirtualization is that it is even faster than trap and emulate as a way of accessing privileged operations from a virtual machine.

The disadvantage is that it simply isn't always available.

For an open-source operating system (like Linux), it is possible for anyone with the necessary skills to make the required paravirtualization changes since the source code is publicly available.

However, this technique doesn't work for proprietary operating systems (like Microsoft Windows) unless the paravirtualization changes are made by the OS vendor--since they are the only ones who have access to the kernel source code.

But even in the case of Linux, not all versions have paravirtualized variants available, and certainly not everyone who uses Linux has the skills necessary to make the changes themselves. So paravirtualiztion is usually done by the vendor of either the OS or the virtualization software.

Even in proprietary operating systems like Windows, there is one area where paravirtualization is used. That is in drivers. Drivers are specialized pieces of software designed to act as an interface to a particular piece of hardware. Drivers are often written by parties other than the OS vendor (e.g. Microsoft). This is needed to allow hardware vendors to provide drivers for their products. The point is that drivers, even those that come from sources other than the OS vendor, run in priv mode. A virtual-machine product can take advantage of this ability to provide drivers. A driver can be provided that uses paravirtualization.

Despite its limitations, paravirtualization has an important role to play. It is, for example a primary method used by Xen.

CMS (the Conversational Monitor System), part of VM/370 was the first operating system to use paravirtualization. All I/O to the console and to virtual disks was done by directly invoking the CP (Control Program) from the virtual kernel.

Paravirtualization is also used by Hyper-V, VMware, and VirtualBox, each of which provide code that is designed to run in selected virtual operating systems.   Microsoft Windows provides device drivers, and Linux provides dynamic kernel modules. Each of these mechanisms can be used to provide paravirtualized code for these systems. For example graphic and network drivers are commonly paravirtualized. Less obvious, but equally important, both memory management and time management are enhanced via paravirtualized modules.

Accessing the Hypervisor: Back Doors

In the case of paravirtualization, as well as for the code that is created by binary translation, there needs to be a way for the code running in the virtual machine to communicate requests to the hypervisor. There are potentially at least six techniques for doing this:

1. A Reserved Instruction: If the creator of the virtual machine has control over the virtual machine language, then a specific instruction can be designated within the machine language to communicate with the hypervisor. For example, VM/370 used the DIAGNOSE

instruction which was part of the original System/360 instruction set but which would otherwise never be used in a virtual machine was used to access CP from the virtual machine.

2. An Invalid Instruction: An instruction known to be invalid in the instruction set can be "co-opted" as a way of invoking the hypervisor. This is somewhat risky in that future enhancements to the instruction set could make the formerly invalid instruction become valid. Microsoft Virtual PC used this technique.

3. Writing to an unused I/O port: This technique assumes that the virtual architecture uses I/O ports; some systems use memory-mapped I/O or other methods that render this technique unavailable. The Intel x86 architecture *does* use I/O ports. Generally, the assignment of specific I/O ports is not defined in the architecture. Rather, a particular machine would have specific I/O devices at specific I/O port addresses. Since the virtual-machine software defines the specific virtual machine, it can control which I/O ports are used for virtual I/O devices and which are reserved for this back-door mechanism. VMware products make use of this technique.

4. Software Interrupt: A back door could simply issue a software interrupt instruction (INT on x86 architecture). The danger with this approach is that the interrupt number could conflict with a software interrupt used by the guest operating system. There is no risk of conflicting with a hardware interrupt for the same reason as #3 above: the virtual machine environment (and thus the set of hardware interrupts) is defined by the hypervisor.

5. Memory Segmentation: In an architecture that supports segmented memory, one memory segment can be reserved for the hypervisor. Code can then jump to an address in the reserved segment. The x86 processor family (but not the x86-64) uses segmented memory. This technique has the advantage of being fast because it does not require any traps or interrupts. VMware uses this technique on 32-bit processors.

6. Reserved Memory: It is possible to “lie” to the operating system and tell it it has less real memory than actually exists. In the case of a virtual machine we are talking about first-level virtual memory--that memory that seems real to the guest operating system, but is actually virtual memory managed by the hypervisor. The “extra” memory that is not included in the indicated memory size can still be addressed. (See the following section on Virtual Memory for more details.) Thus, code (either binary translated or paravirtualized) in the normal address range can make calls or jumps to code in the “extra” memory area. In some cases such code may be sufficient to perform the needed operation. In other cases it may need to use one of the previously referenced methods to access other parts of the Hypervisor.

Handling Interrupts

Interrupts are a hardware mechanism for notifying the operating system (or hypervisor) when events occur that need the OS's attention. For example interrupts are generated by timers, by the completion of I/O operations, and by programming or hardware errors.

If the machine architecture being virtualized includes interrupts, then the hypervisor must emulate them.

On a real machine, the typical sequence of raising an interrupt is as follows:

1. Interrupts are disabled to prevent multiple interrupts from happening at the same time.

2. Some set of machine state (minimally the instruction address and privilege mode) are stored into memory or CPU registers..

3. The address of an interrupt service routine and possibly other state information is made "current" by placing it into CPU registers.

4. Execution continues as normal from the newly specified location and machine state. This causes execution to begin at the location specified in step 3; thus the interrupt service routine begins execution.

Virtual Interrupts

In a real machine these steps are done by hardware. In interrupts to a virtual machine they are done by the software in the hypervisor. Each of the steps listed above can be done by software if we simply insert the word "virtual" at appropriate places in those steps. For example, "*Virtual* interrupts are disabled...."

Real Interrupts

When a real interrupt occurs it can have one of several destinations. It is up to the hypervisor to actually receive the interrupt and determine into which category it falls:

1. It could be an interrupt that should be handled by a virtual machine. The hypervisor may make changes to the details of the interrupt and it then emulates the four steps described above within the context of the intended virtual machine. This is what is described in the previous section, "Virtual Interrupts." This process of sending a real interrupt to a virtual machine is sometimes called "reflecting the interrupt."

2. If the virtual-machine product is running under a host operating system, then the interrupt could be intended for the host. In that case the interrupt is reflected to the host in a manner very similar to that used for a virtual machine. This is an unusual case where the hypervisor emulates a hardware activity to the *host* operating system.

3. It is also possible that an interrupt (for example a timer interrupt) is intended for internal use by the hypervisor. In that case the interrupt is processed by the hypervisor.

Virtual Memory

In general, virtual memory is easy to describe, because it is essentially the same as for any modern operating system. The primary reasons that virtual memory has become ubiquitous are:

(1) it provides isolation. (Each process has a separate set of addresses (or "address

space”.)

(2) It largely solves the problem of “program relocation”. Prior to virtual memory the addresses within a program had to be altered depending on where the program loaded in memory. With virtual memory, a program can always load at the same address in the virtual address space assigned to it.

(3) It provides a simple solution to the problem of a program that is larger than the physical amount of memory available. Prior to the advent of virtual memory, the solution to this problem was largely left to individual programs to solve.

Although most of virtual memory is implemented in software, hardware support is needed in the form of a feature called Dynamic Address Translation (DAT) which translates the address specified by the software (the "virtual" address) to the actual address (the "real" address) where the data is stored. The DAT generally uses a "page table" created by the kernel and stored in memory. Since virtual memory is frequently larger than real memory, not all virtual-memory locations are actually present in real memory. The excess virtual memory is stored on disk. If the hardware references one of these absent virtual addresses, the DAT component generates a "page fault" interrupt which must be handled by the OS kernel. The kernel would usually pick a little-used piece of memory (a "page") and write it out to disk. It would then read in the needed page from disk to memory and alter the tables used by DAT so that it can find the needed page. The failing instruction is then re-executed.

That is essentially what happens in an operating system and it is also what happens in a virtual-machine environment, except that in the case of the virtual-machine environment, the.hypervisor replaces the kernel as the agent handling the page fault and managing the page table.

That would be the end of the story except for one problem: Just as the hypervisor creates a virtual memory for the guest operating system to use, so the guest operating system also creates a second level of virtual memory for its processes to use. Thus there are two levels of virtual memory plus real memory.

The terminology for these levels of memory vary from one virtualization product to another and even from time to time within a product line. The following table gives three sets of terminology for two products. Personally, I like the IBM terminology best as being less ambiguous and more descriptive.

Level of Memory	*Term used by IBM VM/370*	*Old Term used by VMware*	*New Term used by VMware*
Virtual memory created by the guest operating system	2nd Level Virtual Memory	Virtual Memory	gVA (guest virtual address)
Virtual memory created by the hypervisor (seems real to the guest OS)	1st Level Virtual Memory	Real Memory	gPA (guest physical address)
Real hardware memory	Real Memory	Machine Memory	hPA (host physical address)

So there are two page tables: One, created by the guest OS, to translate from 2nd level virtual addresses (to use IBM's terminology) to 1st level virtual addresses, and a second page table, created by the hypervisor, to convert from 1st level virtual addresses to real addresses. Which of these page tables should the DAT hardware use? The answer is, neither. The DAT will actually have to translate from 2nd level virtual addresses to real addresses, so a third page table will need to be created (by the hypervisor) for that purpose. This additional page table is called a "shadow page table".

Whenever the guest OS changes its page table, the hypervisor will have to detect that change and update the shadow page table.

There are two potential mechanisms for detecting OS changes to its page tables. The first method uses memory protection. In the real machine the memory locations that are used for the OS's page table are write-protected so that a trap is generated whenever the guest kernel attempts to write to its page table. The second mechanism is to trap execution of privileged instructions that notify the DAT of page-table changes. The second mechanism might seem to be more efficient, but in fact, such a trap requires scanning the entire page table (which can be quite large) looking for changes.

Actually, either mechanism introduces significant overhead. For this reason several optimizations have been done to reduce that overhead.

In the case of IBM VM/370, one guest operating system, OS/VS1, was modified to not do any paging when run in a virtual machine, thus eliminating one level of virtual memory. This is not a usable general solution, however. OS/VS1 had only one virtual address space, shared by all processes. All modern operating systems have a separate address space for each process. It requires virtual memory (2nd level virtual memory if running in a virtual machine) to implement this feature.

Intel and AMD x86 and x86-64 processors produced in the last few years have an additional hardware feature specifically to support virtual memory in a virtualization environment: These processors directly support two simultaneous page tables, the one created by the guest OS and the other created by the hypervisor. Thus, the DAT hardware directly does both address translations, eliminating the need for the shadow page table.

Virtual I/O Devices

There are many kinds of input/output devices attached to a typical computer and so there are many mechanisms for virtualizing them.

Just as virtual CPUs are in a sense backed by a real CPU, and just as virtual memory is backed by real memory, so virtual I/O devices are usually (but not always) backed by a real I/O device of the same general type.

There are three primary mechanisms for relating a virtual I/O device to a real device:

1. A real device is mapped to the virtual device and used exclusively by the virtual machine to which it is allocated. As an example, VMware allocates the floppy disk drive on the real machine to one virtual machine at a time.

2. A real device is shared by multiple virtual devices. For example a CD-ROM can be

mounted on the real machine and shared among any and all virtual machines.

In addition to the above, it is possible to have "pure virtual" devices that are not associated with any real device. For example, a virtual network device may communicate only with other virtual machines on the same host, so there is no real network device associated with it.

3. A device is “wholly virtual”, i.e. it is emulated entirely in software with no attachment to any real I/O device.

As with virtual memory the language-related virtual machines such as the Java Virtual Machine, use different mechanisms that will be discussed in subsequent sections.

Let's now examine specific classes of I/O devices.

Storage

There are four sub-types of storage:

1. Hard drives

2. Removable magnetic media (floppy disks)

3. Removable optical media (CD-ROM, DVD-ROM)

4. USB Storage

Of these, hard drives are the most important and have the most options.

Most virtual-machine systems support both of the two mechanisms mentioned above: One can either dedicate a real disk drive to a virtual machine, or a real drive can be divided into discrete pieces and each piece becomes a virtual disk drive in the virtual machine.

For the other types of devices, a real device is usually attached to the virtual machine. If the device is read-only (e.g. a CD-ROM) then the device can be shared between multiple virtual machines. If, however, it is read/write, then it is usually dedicated to a single virtual machine at a time. As an alternative, the image of a removable disk can be stored in a file on a hard disk. For example, a CD-ROM can be copied to a disk file in a format known as an ISO image. That ISO image (stored as a file on a hard drive on the real machine) can be attached to a virtual machine as a virtual CD-ROM.

USB devices are generally attached to whichever virtual machine is “in the foreground” at the time the device is plugged into the host computer.

Networking

A real computer usually has a Network Interface Card (NIC). By analogy then, a virtual machine will have a virtual NIC. But this is not sufficient. How is the virtual NIC connected to a network?

In a real network, computers are connected to either a hub, a switch, or a router. So the virtual NIC needs to be connected to a virtual hub, switch, or router. It's typically a virtual switch. The switch can connect all of the virtual machines on a given host. The virtual switch can also be connected to the host computer and to a real NIC in order to gain connectivity to a real network.

Printers

Most operating systems use a technique called "spooling" to share a single printer among multiple processes running on that system. It is common for a hypervisor to use the same technique. In spooling, a program (or virtual machine) writes its printer output to a temporary disk file and once the program is finished, and once the printer becomes available, the disk file is copied to the printer.

Human Interface Devices

Human interface devices include the keyboard, mouse, and screen.

Keyboard Focus

In a typical operating system, one window is given the "focus" meaning that that process has control of the keyboard and mouse. The hypervisor can make similar allocations. In fact, if the hypervisor is running under a host operating system, it is generally the windowing system on the host that decides who has the focus.

Graphic Display Adapters

Display adapters fall into the category of human interface devices. But they deserve specific mention because of their complexity and also their unique design qualities. In the days of VM/370, the "display" device could well have been a sheet of paper in a typewriter terminal. Even if it was a video display monitor, it was still a character-only device, and it was updated by sending a simple stream of characters one at a time. This was easy to virtualize. Each virtual machine had a dedicated terminal device for a console.

The Apple Macintosh was the first widely popular personal computer to use a graphical user interface. It was black and white and the interface, while more complex than the previous character-mode displays was nevertheless fairly simple. The screen was divided into small dots, called pixels. The image on the screen was created by a piece of dedicated memory where every bit in the memory corresponded to a pixel on the screen: if the bit was 1, the pixel was white; if the bit was 0, the pixel was black. The bits that made up all of the images on the screen, from lines, to "round-rect" buttons, to text, to drawings, was computed by software.

While color added more bits per pixel, the principle was the same: the software computed the right bit patterns in the video memory to create the desired visual image. This was very compute intensive, and so more and more intelligence has been moved into the graphic interface card to remove more and more overhead from the CPU. In fact the graphic interface has been re-christened as a GPU--graphic processing unit. It has become a processor in its own right, and in some cases actually uses standard processor chips. More often, however, the processor chips used are proprietary, and their design is not made public. Instead, the graphics-card vendor provides driver software for supported operating systems.

The graphical video adapter for the first IBM PC was called CGA--Color Graphics Adaptor. This evolved into EGA, and then VGA, each with higher resolution and more colors. But each of these had a standard software interface. Then came SVGA, which is the "standard" today. However, it is no standard at all since each manufacturer provides its own proprietary software interface. The only requirement is that these adaptors support the older VGA standard.

Virtualizing these complex--and ever evolving--SVGA graphical interfaces is a significant challenge. The problem has generally been addressed in two ways.

(1) virtual-machine software providers fall back to the VGA interface, the last standardized graphic interface. But this has a low resolution (640x480 pixels) and a small number of colors (256).

(2) The other solution is for the virtual-machine vendor to provide a paravirtualized graphics driver, so that the driver doesn't have to support a real GPU. This allows for dividing the labor between the driver running in the guest, the hypervisor, and the driver running in the host, in any fashion that the designers wish. [9]

Aside from the complexities of emulating the graphics hardware, there is another question: who owns the screen? In a normal operating-system environment, some component of the operating system owns the screen and divides it up into task bars, menu bars, etc., and, mostly, windows, roughly one window per application.

But in a virtual-machine environment there are multiple operating systems each of which "thinks" it "owns" the screen. There are several solutions to this problem. Often, all of them are available at the user's selection:

1. The screen can be owned by one of the guest operating systems. In that situation the visual appearance is the same as if that operating system were running on the real hardware.

2. The screen can be owned by the virtual-machine software. In that case the screen could theoretically be divided among the running virtual machines. I say "theoretically" because I'm aware of no product that actually does things that way. What does happen, in VMware, for example, is that the top and bottom edges of the screen contain controls for selecting and monitoring virtual machines, and the center of the screen is a slightly smaller version of the real screen and is managed by one (user-selected) virtual machine's operating system.

3. The screen can be owned by the host operating system. There are two variations of this scheme. In one variation, each virtual machine's screen is shown in a window on the host's screen. The other variation is more complex and more difficult to implement, but potentially more useful. In this second variation, individual windows from one or more guest OS's are intermingled on the same screen. This gives the user a kind of hyper desktop containing windows from the applications of multiple (both host and guest) OS's

Handling Real I/O in a Virtual-machine Environment

At some point, virtual I/O must be turned into operations against a real I/O device. From where is this real I/O initiated? There are actually several possibilities:

1. The I/O operation can be handled directly by the hypervisor. Since this component has direct control of interrupts and is running in privileged mode it certainly has the ability to do this.

2. If the hypervisor is running within a host operating system, then the I/O operations can be passed directly to the host operating system.

3. Some solutions provide a "special" guest operating system whose role it is to perform real I/O operations on behalf of the other virtual machines.

Specific Implementations

In this section we will select some of the virtual-machine systems described in the history section to see how they made use of the techniques in the previous section to form a specific solution.

VM/370

VM/370 implemented IBM System/370 virtual machines on a System/370 host. The hypervisor was implemented in the CP (Control Program) component. VM/370 ran on the "bare metal"; it did not use a host operating system.

Virtual CPU

VM/370 is the type of system where the architecture of the virtual machine matches the architecture of the real machine. Recall that this means that only privileged instructions need to be emulated. The technique that was used was "trap and emulate". Every reference to a privileged machine state required a privileged instruction. Executing such an instruction from a virtual kernel not running in kernel mode, generated a "program check" interrupt with an interrupt code indicating "privileged operation exception". The CP would then handle this exception and emulate the trapped instruction.

The System/370 upon which VM/370 ran was a microcoded system. IBM eventually added specific microcode features, collectively known as "Virtual Machine Assist" (VMA) to make frequent operations associated with a virtual machine more efficient.

A few guest operating systems (particularly CMS) were modified to use paravirtualized I/O as a further performance enhancement.

Virtual I/O

A virtual disk device for a VM/370 virtual machine could either be a dedicated disk drive or a pre-allocated section of a disk, known as a mini-disk or mdisk. The space for these mdisks had to be hand-allocated by the system administrator.

The console of a real System/370 was a locally-attached typewriter device. The console for a virtual machine was a network-attached terminal.

VM/370 was created in the days of batch operating systems and punched cards. VM/370 supported virtual card readers and card punches using the same spooling techniques used for printers.

Real I/O performed on behalf of a virtual machine was done by CP, the control program which acted as both the hypervisor and the host OS.

Java Virtual Machine

The Java Virtual Machine runs on a wide variety of host environments and creates virtual machines that execute the ficticious machine language known as "bytecode". The hypervisor is the JVM itself.

Virtual CPU

The Java Virtual Machine executes machine instructions from a fictitious machine language. Therefore every virtual machine instruction must somehow be translated to one or more real instructions. Early JVMs used interpretation for this purpose. New JVMs use binary translation in the form of JIT or "Just in Time" translation. The amount of code that is translated and cached at one time is one Java method.

Virtual Memory

The Java Virtual Machine has no concept of virtual memory. In fact the Java language has no direct mechanism for allocating memory, other than the ability to create objects and arrays. The memory that is used is simply the host operating system's virtual memory that is made available by the host OS to the process in which the JVM is running.

Virtual I/O

In a similar fashion to the JVM's use of memory, I/O devices are managed by the host operating system. Files (and devices if they appear as files in the host OS) are made available to the Java program via Java I/O classes that are implemented as interfaces to the host OS file system.

VMware

VMware offers a variety of products but they all share a common architecture that will be described here. All of these products implement Intel x86 or x86-64 virtual machines on the same class of host computers. If the host is x86-64 hardware, then both x86 (i.e. 32-bit) and x86-64 (64-bit) virtual machines can be run. Some VMware products run on a host operating system which may be either Windows, Linux or the Mac OS. Others run on the "bare metal" by running their own kernel, the VMKernel.

Virtual CPU

Traditionally VMware products use binary translation as the means of CPU virtualization In fact, VMware pioneered this technique for x86 virtualization and in effect broke away from the general model of trap-and-emulate which had been invented by IBM for CP-40, CP-67 and VM/370.

VMware calls it's hypervisor the "Virtual-Machine Monitor".

VMware products use Just in Time (JIT) binary translation to convert privileged instructions into alternate code (generally backdoor calls to the hypervisor). The amount of code that is translated at one time is from the current instruction to the next instruction that can or will cause a transfer of control (i.e. a jump, subroutine-call, or software interrupt).

Newer Intel and AMD processor have hardware virtualization support, which means that they can be placed in a mode where all privileged-state access *will* be trapped, making trap-and-emulate feasible. Surprisingly, the pure software method of binary translation can sometimes be faster, since the "trap" in trap-and-emulate (and, in particular, the set of instructions necessary to service that trap) is an expensive operation. [7]

VMWare also supplies paravirtualized drivers for many of the most popular guest operating systems. This collection of components is known collectively as “VMWare Tools.”

Virtual Memory

VMware must support the three-tier model of memory described previously. VMware software generally supports maintenance of the shadow page table by write-protecting the pages that contain the guest OS's page tables. Newer processors have a hardware mechanism for handling multiple levels of virtual storage by supporting two page-table pointers and resolving in hardware the double translation (second level to first, and first to real addresses) .

Virtual I/O

For "hosted" products--those that run in a host operating system--each virtual machine has a corresponding process, called the VM Extension (VMX) process that runs in the host operating system. It is this VMX process that communicates with the host operating system to handle I/O requests.

For ESX, the VMware product that runs directly on the hardware--without a host OS-- the hypervisor communicates I/O requests to the VMKernal which contains drivers to handle I/O to real devices.

VMware products handle the range of I/O devices supported by the x86 architecture:

Virtual hard drives can be emulated using files in the host operating system, or can be attached to real hard drives.

Virtual optical disks and floppy disks can be attached to real devices or can be associated with ISO images, (copies of the contents of a CD, DVD, or floppy disk, stored in a file on a magnetic disk).

For networking, VMware creates both virtual NICs (Network Interface Cards) and virtual switches that are used to connect the virtual machines on a given host so that they can communicate among one another and with the host. To connect this virtual switch to a real network, three options are provided:

(1) Packets from the virtual network are kept local to the host machine. In this case a virtual machine would have no direct access to a real network via the virtual network.

(2) The virtual network is managed via NAT (Network Address Translation). In this option virtual machines have local IP addresses so they can communicate among themselves. Packets destined for a real network have their packets altered to use the IP address of the host computer. (Consumer broadband network routers use the same NAT technique to share multiple home computers with a single IP address provided by the Internet service provider.)

(3) Packets from a virtual machine are forwarded intact to the real NIC on the host. This causes the real machine to present multiple MAC (Media Access Control) addresses to the real network (one MAC address for the real machine and one additional address for each virtual machine). This is a simple problem for outgoing packets, since the MAC address is supplied by the network software. For incoming packets, the real-machine's NIC is placed in "promiscuous mode" meaning that it will read in packets with any destination address. (Normally the NIC would only present to the software packets that had the destination address associated with that NIC.) It is up to the hypervisor to distribute incoming packets to either the network software on the host or to forward them to the virtual NIC in the virtual machine, depending on the value of the destination MAC address.

Keyboard, mouse, and video are handled by interfacing to the underlying windowing system running on the host OS.

VMware products are designed to work with unmodified guest OSes. However VMware also supplies a collection of components, mostly I/O drivers, that can be installed in a guest OS to improve performance and functionality by using paravirtualization.

How real I/O is handled depends on the product. For products designed to run in a host OS, the I/O operations are given to the host OS to perform. For products that run on the bare metal, there are a kernel and drivers, supplied by VMware, that handle I/O for the virtual machines.

Xen

Xen is an open-source virtualization product. It creates x86 and x86-64 virtual machines on the same family of host systems.

Xen is a cross between a bare-metal hypervisor and a hosted hypervisor. The hypervisor runs on the bare metal, but it uses one Linux operating system to provide real I/O operations. However, this OS is run as a guest rather than the host of the hypervisor.

Virtual CPU

Xen uses a combination of paravirtualization (primarily for Linux guests) and hardware virtualization support (for any guest OS).

Virtual Memory

In paravirtualized virtual machines in Xen the guest OS requests blocks of memory from the hypervisor. Writes to that memory are trapped and checked by the hypervisor.

In non-paravirtualized virtual machines in Xen, virtualization hardware support provides for two levels of virtual memory as described previously.

Virtual I/O

As with memory and CPU, there are two implementations of virtual I/O under Xen: In paravirtualized virtual machines, there is a set of interfaces provided by the hypervisor for doing I/O, and drivers in the guest OS must be modified to use those interfaces. In non-paravirtualizxed vitual machines, native drivers are supported by trapping the privileged operations that drivers use to interface to an I/O controller.

Real I/O is handled by "Domain 0" a special virtual machine that controls the other virtual machines. This domain 0 machine then makes standard I/O calls.

Microsoft Hyper-V

The original implementation of Microsoft virtualization was done by Conectrix, a third-party company that Microsoft acquired. More recently, Microsoft has a new virtualization product, called Hyper-V. [10]

The implementation of Hyper-V is not documented for public consumption in any great detail. What has been publicly described indicates an implementation similar to Xen.

However, the terminology Microsoft uses is different from Xen (and from any other virtualization product):

- What Xen calls "Domain 0," Hyper-V calls the "Root Partition". This entity acts as the interface to the host operating system for doing I/O and other system functions.
- Each virtual machine runs in a "child partition".
- Paravirtualized virtual-machine kernels are said to be "enlightened". They communicate with the root partition and the hypervisor as well as with other enlightened kernels in other partitions, using a facility known as "VMBus."
- Hyper-V can also run unmodified OSes which are said to be "unenlightened." Such VMs require hardware virtualization support in order to implement trap-and-emulate mechanisms.

Virtual CPU

"Enlightened" guest OSes use paravirtualization. This means that the virtual Kernel must be built specifically to run in a Hyper-V virtual machine and it must know how to communicate with the hypervisor. Microsoft provides interface libraries for Windows and Linux guest OSes. Unlike Xen, Hyper-V can support paravirtualized Windows OSes since Microsoft has access to the Windows kernel source code in order to make the needed paravirtualilzation changes.

Hyper-V can support hardware virtualization as provided by newer Intel and AMD processors. This means it can support unmodified OSes.

Virtual Memory

In the enlightened OSes under Hyper-V, the guest OS requests blocks of memory by communicating with the hypervisor.

For unenlightened OSes, virtualization hardware support provides for two levels of virtual memory as described previously.

Virtual I/O

As with memory and CPU, there are two implementations of virtual I/O under Hyper-V: In an enlightened OS, there is a set of interfaces provided by Microsoft for doing I/O, and drivers in the guest OS must be modified to use those interfaces. In unenligtened OSes, native drivers are supported by trapping the privileged operations that drivers use to interface to an I/O controller.

Real I/O is handled by the "Root Partition."

VirtualBox

Virtual CPU

VirtualBox seems to utilize every one of the software virtualization techniques mentioned in this paper. [13] (In fact, "Patching in Place" was added to this paper because of its use in VirtualBox.) VirtualBox also supports both Intel and AMD hardware virtualization. It refers to its implementation of paravirtualized components as "guest additions".

Virtual Memory

The User's Manual for VirtualBox has an entire chapter on internals, and describes CPU virtualization in detail. It says less about software implementation of the three-level storage described in this paper (which it refers to as "nested paging".)

Virtual I/O

As with virtual memory, details are sparse. In terms of virtual I/O features almost everything described above VMware's virtual I/O also applies to VirtualBox with the addtion that it supports multiple formats for virtual disk images (including VMware's). It also supports two additional networking modes: host-only mode and UDP-tunneling.

KVM

Virtual CPU

KVM relies on CPU hardware virtualization.

Virtual Memory

KVM relies on hardware virtualization support for virtual memory.

Virtual I/O

KVM basically punts I/O support back to user space. (KVM operates entirely in kernel space). It is up to the user-mode client to handle I/O. (In practice KVM is usually run under QEMU which supplies I/O emulation.)

QEMU

Virtual CPU

QEMU supports two modes of CPU virtualization:

- It has an engine that supports emulation via JIT binary translation. In this mode QEMU has a library of front-ends that translate the virtual-machine code into a canonical intermediate form. It also has a library of back-ends that translates the intermediate form to the machine code of the host machine.
- It also has been adapted to use KVM for supported architectures where the host and guest architecture is the same.

Virtual Memory

QEMU allocates a (large) block of memory from the process it is running in to supply the "real" memory to the virtual machine.

Virtual I/O

QEMU uses the host operating system to supply I/O for the virtual machine.

References

Note: Most of the material in this paper is derived directly from my own experience from working with this software over a period of over thirty years. I have relied on the following sources for additional detail.

[1] Creasy, R. J., "The origin of the VM/370 time-sharing system", *IBM Journal of Research & Development*, Vol. 25, No. 5 (September 1981),

[2] Otten, Hans, "UCSD P-System: History", http://pascal.retro8bits.com/UCSDhistory.html

[3] Byous, Jon, "Java Technology: The Early Years", http://java.sun.com/features/1998/05/birthday.html

[4] "VMware Leadership", http://www.vmware.com/company/leadership.html

[5] Knuth, Gabe, "A Brief History of Xen and XenSource", http://www.brianmadden.com/blogs/gabeknuth/archive/2007/08/16/a-brief-history-of-xen-and-xensource.aspx

[6] Riddel, Jonathan, "History of the Pentium Processor", http://jriddell.org/programs/essays/x86-essay.html

[7] Adams, Keith and Ole Agesen, "A Comparison of Software and Hardware Techniques for

x86 Virtualization," *Procedings of ASPLOS'06.*

[8] "What is a Virtual Machine", http://www.vmware.com/technology/virtual-machine.html

[9] Dowty, Micah, and Jeremy Sugerman, "GPU Virtualization on VMware's Hosted I/O Architecture," *Operating Systems Review,* Volume 43, Number 3, July 2009

[10] "Hyper-V Architecture," http://msdn.microsoft.com/en-us/library/cc768520(BTS.10).aspx

[11] "Microsoft Buy to Boost Server Efforts", http://news.cnet.com/Microsoft-buy-to-boost-server-efforts/2100-1001_3-985149.html

[12] "z/VM", http://www.vm.ibm.com/

[13] “Oracle VirtualBox User Manual”, https://www.virtualbox.org/manual/. This manual represents some of the best documentation about a virtualization product that I have run across.

[14] The NIST Definition of Cloud Computing, http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf

For an excellent source of additional detail about virtualization, and specifically that from VMware, see *Operating Systems Review,* Volume 44, number 4, December 2010. This entire issue is dedicated to articles about VMware. See particularly the first article, "The Evolution of an x86 Virtual Machine Monitor" by Ole Agesen, Alex Garthwaite, Jeffery Sheldon, and Pratap Subrahmanyam.