Virtual Machines

When you finish reading this chapter you should be able to:

- Briefly describe the ideal program development environment.
- Distinguish between virtual and real.
- Briefly explain the virtual machine concept.
- Explain the term transparent.
- Explain how VM helps to solve the software conversion problem.
- Explain how the virtual machine concept generates leverage to significantly increase the potential number of concurrent application routines.
- Sketch the components of a virtual machine.
- Sketch the contents of real memory on a computer running under VM.
- Explain how fixed memory locations are implemented under VM.
- Explain how the real control program simulates interrupts for the virtual machines.
- Explain why CMS is considered an important component of IBM’s VM/SP.
- Briefly explain how the control program manages the real processor’s time.
- Explain how memory is managed under VM.
- Briefly explain paging under VM.
- Distinguish between a real peripheral and a virtual peripheral.
- Explain how the control program keeps track of its various virtual machine environments.
- Explain how interrupts are handled under VM.
- Explain how (and why) the control program traps privileged instructions.
Before you begin reading, be sure you understand virtual memory (Chapter 6), traditional IBM mainframe principles of operation (Chapter 17), and MVS internals (Chapter 18).

## Operating System Development

Early, first-generation computers did not have operating systems. Primarily scientific machines, they were dedicated to a single user who wrote and tested programs at the console. As a result, response time was excellent, but given the cost of the equipment, the single-user mode of operation was economically unsound.

The first operating systems supported efficient, serial-batch job-to-job transition. Typically, responsibility for running these early systems was assigned to a professional operator. Instead of working at the console, programmers prepared job decks on punched cards, submitted them to the operator, and returned a few hours later to pick up the results. Most multiprogramming operation systems were designed with such batch processing in mind.

The objective of batch processing is machine efficiency; ideally, there is always at least one job waiting for the computer, so the computer is never idle. Unfortunately, a batch system is not an efficient program development environment. Imagine writing a program on coding sheets, waiting a day or two for the punched cards to come back from the key-punching center, visually checking the cards, submitting the deck for compilation, waiting several hours (sometimes, a day or more) for the results, correcting a few cards, resubmitting the deck, waiting a few more hours, and so on. Programming requires concentration. With brief bursts of activity separated by lengthy wait times, it is difficult to write a good program on a batch system.

Time-sharing was an alternative. On a time-sharing system, programs are developed interactively, which is far more efficient. Unfortunately, most early time-sharing systems restricted a programmer’s access to system resources, and many supported only a few languages such as BASIC and APL, making it difficult to adequately write and test large programs. Consequently, most business applications continued to be developed in a batch environment.

Modern personal computers have brought us full circle. Once again, it is possible to dedicate a complete machine and all its resources to a single programmer. However, although personal computers are at least as powerful as a typical first-generation machine, the definition of computing power has changed. Today’s mainframes contain millions of bytes of main storage, execute millions of instructions per second, and support scores of secondary storage devices. Personal computers are simply not powerful enough to support testing and debugging mainframe applications.

The ideal program development environment combines the interactive nature of a personal computer with the power of a mainframe. The programmer has access to a full set of peripherals, megabytes of memory, mainframe computing speed, and a mainframe’s full, rich instruction set. Such environments can be supported on a modern virtual machine system.
The Virtual Machine Concept

Start with a full-featured mainframe. Share its resources among several concurrent users. If those users occupy partitions, regions, or workspaces, you have a traditional multiprogramming or time-sharing system. Take the idea a step further. Instead of simply allocating each application routine some memory and running it directly under the operating system, simulate several imaginary computers on that real computer (Figure 19.1). Assign each virtual machine its own virtual operating system and its own virtual peripherals. Traditionally, application routines are multiprogrammed. The virtual machine concept calls for multiprogramming operating systems.

Each virtual machine has its own virtual operating system, its own virtual memory, and its own virtual peripherals. Because all the virtual machines run on the same real computer, their access to facilities is limited only by the facilities of the real machine. Thus, each virtual machine has access to megabytes of storage and scores of peripherals, and can execute millions of instructions per second. Because they share a single real computer, program development can take place on one virtual machine in interactive mode, while pro-

---

**Figure 19.1**
The virtual machine concept implies multiprogramming at the operating system level.
duction applications run on another virtual machine under a traditional multiprogram-
ming operating system.

To the user, the virtual machine is the computer. The details associated with the real
machine are transparent, hidden by the facilities of the “real” operating system. Thus,
much as a time-sharing user can ignore other, concurrent users and imagine that he or she
directly controls the computer, a virtual machine user can ignore other virtual machines.

A UNIX user (Chapter 16) can visualize an image running on a personal psuedo-com-
puter. Details, such as the number of users sharing a text segment, dispatching, swapping,
and peripheral device linkage, can be ignored because the operating system makes them
transparent. The UNIX psuedo-computer is a virtual machine. This concept is one reason
why professional programmers consider UNIX such a friendly program development envi-
ronment.

IBM has implemented the virtual machine concept under VM/SP (Virtual Machine/System Product). Chapters 17 and 18 discussed traditional IBM principles of
operation and the MVS family of operating systems. In this chapter you'll study VM/SP,
building on the base established in the last two chapters.

### VM/SP

In 1964, IBM announced System/360 and ushered in the computer’s third generation.
While other manufactures sold comparable machines before 1964, IBM’s decision to
switch from discrete transistors to integrated circuits legitimized the newer technology.

The change was not entirely positive, however. Third-generation technology differed
radically from second-generation technology, supporting advanced operating systems and
enhanced instruction sets. Thus, most second-generation programs were rendered obso-
lete. The manager of a second-generation computing center was faced with three almost
equally unacceptable choices: keep the old, obsolete, inefficient hardware; emulate a sec-
dond-generation computer on a third-generation machine (thus losing many of the advan-
tages of the new hardware); or rewrite existing programs.

Several new operating systems were developed to support the IBM System/360 and its
successors (Figure 19.2). Initially, DOS proved quite popular, but IBM’s mainstream oper-
ating system was OS. OS/MFT and OS/MVT were released in the 1960s. By the mid-
1970s, they had evolved, respectively, to VS1 and VS2; eventually, VS2 became MVS.

While significant improvements were made, DOS was a dead end, and IBM has consis-
tently urged its users to migrate toward its OS products. Many DOS users were reluctant
to change, however. Programs written under DOS and OS are different, particularly when
they communicate with an external device. Consequently, converting a DOS program to an
equivalent OS program, even in the same source language, means, at a minimum, changing
all the input and output routines. A typical business data-processing shop might have hun-
dreds of such programs, and converting them is a significant expense. Thus, IBM contin-
ued to support DOS.

If all IBM applications had been written under OS, upgrading to a new, more pow-
erful operating system would be relatively easy, but DOS proved surprisingly popular.
Consequently, IBM faced a problem. Clearly, a new operating system was needed.
Maintaining compatibility with two essentially incompatible operating systems, DOS and OS, was considered essential. VM emerged as a solution.

Under IBM’s VM, the real computer’s resources are managed by a high-level operating system called the control program or CP (Figure 19.3). Normally, application routines run under the operating system. With VM, operating systems are managed by the control program, and application routines run under those virtual operating systems.

The result is considerable leverage. For example, on a normal VS1 system up to 15 application routines share memory with the operating system. With VM, up to 15 concurrent virtual operating systems can each manage up to 15 application routines, yielding

**Figure 19.2**
Since IBM’s System/360 was announced, several operating systems have evolved to support it and its successors.

**Figure 19.3**
Under VM, the real computer is managed by a control program. Other virtual operating systems run under CP, managing their own application routines.
225 potential concurrent programs! It is also possible to run a virtual control program under a real control program (Figure 19.4). Imagine 15 copies of CP, each of which manages 15 virtual operating systems. That’s 225 *operating systems*. Now, picture each of them managing 15 application routines. That’s leverage!

Each virtual operating system manages a virtual machine and controls its own virtual memory and **virtual peripheral devices**. A virtual machine is functionally equivalent to a real machine. The control program simulates a real computer on each virtual machine; consequently, the virtual operating systems can function without change. Any DOS- or OS-based operating system can run under VM.

For example, consider the problem faced by a DOS shop that has outgrown its system. A faster computer with more memory capable of supporting more concurrent application routines is needed, but hundreds of DOS programs exist, and revising all of them is clearly out of the question. With VM, the programs need not be revised. Instead, two virtual copies of DOS/VSE can be loaded. Each one can support up to five concurrent application routines, thus doubling the system’s capacity. Additionally, those routines can run on a larger, faster, more powerful computer.

![Diagram](image.png)

**Figure 19.4**

It is even possible to run another control program under CP.
VM’s Structure

Under VM, the real computer’s resources are managed by the control program (Figure 19.5). Real memory starts with several fixed locations—old PSWs, new PSWs, the channel address word, the channel status word, and so on. Next comes the resident portion of CP; it holds key tables and control blocks plus routines that dispatch virtual machines, manage real memory, control paging, handle interrupts, and so on. Some CP modules are pageable; space for them comes next. Finally, the remaining real memory is divided into real page frames. Pages are swapped between the real page frames and the external paging devices associated with one or more virtual machines.

Visualize each virtual machine as though it were a complete computer in its own right. Its virtual memory begins with a set of virtual fixed memory locations; in other words, each virtual operating system has its own old and new PSWs, its own channel command word, and its own channel status word. Following the fixed locations is the virtual operating system itself. Remaining virtual memory holds virtual program partitions or regions.

On every operating system you’ve studied to this point, key modules and control fields were stored in fixed memory locations. Note (Figure 19.5) that the virtual operating systems are subject to paging. Consequently, a virtual operating system’s modules will not occupy the expected, real, absolute addresses. Instead, a virtual operating system runs

---

**Figure 19.5**
The basic structure of a VM system.
under the control program in much the same way an application routine runs under a traditional operating system.

For example, consider the fixed memory locations. The real computer has a set of fixed locations that start at real address 0. Each virtual machine has its own set of fixed locations that start at its virtual address 0. When an interrupt is sensed, which PSWs are switched? The real ones. Interrupts are sensed by hardware, and, as far as hardware is concerned, there is only one real computer. Once PSWs have been switched, the control program’s interrupt handler routine is activated. If necessary, CP can then switch (through software) the appropriate virtual machine’s PSWs, thus simulating the interrupt at the virtual machine level.

The real control program does not concern itself with the application routines. Each virtual operating system is responsible for managing its own processor time, virtual memory space, and virtual peripheral devices. CP communicates with the hardware, simulates the results for the virtual machines, protects the virtual machines from each other, and allows the virtual operating systems to handle the details.

CMS

CMS, the conversational monitor system, is an interactive, single-user operating system that supports program development. It’s a key part of VM, and thus deserves a more detailed description.

Traditional operating systems are not very good for program development. Multiprogramming implies batch processing. Time-sharing systems are interactive, but most limit the programmer to a small subset of a mainframe’s resources. An ideal program development environment would combine the interactive nature of a personal computer and the resources of a large mainframe. Running as a virtual operating system under CP, CMS does exactly that (Figure 19.6).

CMS simulates a personal computer. The user’s terminal acts as a console. Each user has his or her own CMS virtual machine, complete with virtual memory, virtual batch devices (a reader, a printer, and so on), and virtual disk space. Because it runs under CP, a CMS virtual machine’s facilities are limited only by the mainframe’s facilities. Each user has access to a variety of compilers, a linkage editor, utility routines, and a spooler. Additionally, users can assign work to the CMS batch facility, and thus run two or more independent tasks in parallel. CMS supports a powerful, interactive program development environment.

The Control Program (CP)

The control program (CP) manages the system.
Processor Management

The control program uses a time-slicing algorithm to manage the real processor’s time. A virtual machine is given control of the processor for a fixed amount of time. When the time slice expires, CP passes control to another virtual machine. Interactive virtual machines (for example, CMS users) are assigned frequent but brief time slices. Noninteractive virtual machines (for example, a DOS/VSE system running accounting applications) are assigned fewer but longer time slices. Each virtual operating system manages its own time, shifting from application routine to application routine by responding to interrupts or by implementing a secondary time-slicing algorithm. Thus, each virtual operating system “thinks” it controls access to the processor.

Memory Management

VM can be implemented only on computers that support dynamic address translation. Memory space is managed using segmentation and paging techniques, with 64K-byte segments divided into 4K pages. A demand paging algorithm controls swapping between real memory and one or more external paging devices. The control program maintains a separate set of page frame tables and a separate set of paging and segmentation tables for each virtual machine.

Paging can be initiated by the control program or by one of the virtual operating systems. If initiated by CP, paging is transparent to the virtual machine. As far as the control program is concerned, a paging request coming from a virtual operating system is an I/O operation.

Figure 19.6

CMS is an interactive, single-user operating system that controls a virtual machine under VM.
With both virtual and real operating systems initiating paging, a VM system can appear confusing. Sometimes a simple visualization helps. Imagine a page of data stored on disk. Mentally transfer the page into a virtual machine’s virtual memory (Figure 19.7). Next, move it to the real machine’s virtual memory. Finally, transfer the page to real memory. Note that a single computer’s real memory supports a much larger “real” virtual memory, which, in turn, supports several “virtual” virtual memories. That’s the source of VM’s leverage.

Of course, no computer actually performs all those data transfer operations (that would be terribly inefficient). Instead, pages are swapped between an external paging device and the real page frames, and an algorithm is used to translate virtual addresses to real. For example, imagine starting with a relative address in a virtual machine’s virtual memory. Adding the base register and the displacement yields an “absolute” virtual address. Passing that virtual address through the virtual machine’s segmentation and paging tables yields an absolute address in the real machine’s virtual memory. Finally, passing that address through the control program’s segmentation and paging tables yields a real absolute address. Various mathematical techniques and page address registers can help streamline these computations.

**Managing Peripheral Devices**

Each virtual machine has a set of virtual peripheral devices. All *real* peripherals are controlled by the control program, CP, however. Some devices are nonsharable. For example,
the user’s terminal is treated as a console dedicated to a specific virtual machine and thus becomes the source of that machine’s operator commands. Other nonsharable devices, such as a reader, a printer, and a punch (a source of machine-readable output), are simulated through spooling. Magnetic tapes are generally dedicated to a single virtual machine upon request.

Sharable devices, such as disk, are supported through the minidisk concept. When a user logs onto CP, the main control program generates the appropriate virtual machine. For each disk file requested, CP allocates several tracks or cylinders to the virtual machine. To the control program, those disk extents are files; the virtual machine, however, sees them as dedicated independent “mini” disk packs.

Under CMS, selected minidisks can be shared by several users. For example, in a university system, the instructor and each student might have a private minidisk. An additional minidisk is shared by the instructor and the students. The instructor has read/write access; the students have read-only access; the shared minidisk serves to pass assignments and common code (for example, data) to the students.

Principles of Operation

The control program maintains a system directory with one entry for each virtual machine (Figure 19.8). The entry completely describes the virtual machine environment and includes such data as a list of virtual peripherals, the virtual storage size, valid logon IDs and passwords, accounting data, user priorities, and each user’s command class. Valid command classes are listed in Figure 19.9; they limit the commands a given user is authorized to issue.

A user logs onto the control program. CP then checks its system directory, verifies the user number and password, and generates the virtual machine environment. The user can then IPL the appropriate operating system and begin working.

When an interrupt is sensed, control is passed to an interrupt handling routine in the control program. Interrupts, remember, are sensed by hardware, which switches PSWs
stored in fixed memory locations. Since only CP controls real memory, it follows that the addresses of CP’s interrupt handling routines are stored in the real new PSW fields. Consequently, a CP routine gets control following any interrupt.

Normally, the control program passes the interrupt to the virtual operating system by copying the appropriate old PSW’s contents into the virtual machine’s old PSW field, and then loading the virtual machine’s new PSW. Other relevant fields, such as the channel status word or channel address word, are also copied if necessary. When the virtual operating system’s interrupt handler gets control, its fixed memory locations contain exactly what they would have held had its virtual machine sensed the interrupt. Thus, as far as the virtual machine is concerned, the control program’s manipulations are transparent.

Privileged instructions, such as SIO, are used to start or control physical I/O operations. Because a virtual machine has no physical peripherals (only virtual, imaginary peripherals), it cannot be allowed to directly control physical I/O. Consequently, the control program traps privileged instructions and performs the requested physical I/O operations itself. It then simulates the results and returns control to the appropriate virtual machine.

Virtual operating systems run under CP in the control program’s problem state. (Normally, an operating system runs in supervisory state.) Privileged instructions are illegal in problem state; they generate program interrupts. All interrupts, remember, result in a transfer of control to a CP interrupt handler routine. If the interrupt cause code indicates anything but a privileged instruction, CP passes the interrupt to the virtual machine’s operating system. Otherwise, it reads the privileged instruction and performs the operation.

Each virtual device assigned to a virtual machine has its own virtual channel/device address. For each virtual machine, the control program maintains a list of virtual devices and their real channel/device equivalents. The virtual peripheral device address can be any valid channel/device address; it need not match the real device address. When an I/O operation is requested, the control program extracts the virtual channel/device address, finds the associated real channel/device address in the table, substitutes it, and performs the operation. Results are then reported back to the virtual operating system using the virtual address.

Does this mean that application programs on a virtual machine can issue their own privileged instructions? No. Each virtual operating system has its own problem state bit. When it issues a privileged instruction, a program interrupt is generated because the vir-
ual operating system runs in the real operating system’s problem state. The control program checks the virtual machine’s problem state bit. If it’s 0 (supervisor state), the virtual operating system must have issued the privileged instruction, and that’s legal. If, however, the problem state bit is 1, an application routine issued the privileged instruction, and that’s illegal. Thus, the interrupt is passed back to the virtual operating system for handling.

Advantages and Disadvantages

The virtual machine concept solves a number of problems. It supports a program development environment that combines access to the full power of a mainframe with the response of a personal computer. It provides an efficient means of increasing the number of levels of multiprogramming while protecting customers’ software investments. Compatibility with existing operating systems is a real key, because it allows a customer to move up to more powerful hardware without changing operating environments.

VM is particularly valuable for testing. For example, with VM, it is possible to test an operating system without dedicating a computer to the task (and thus postponing other work). When application routines are modified, the old and new versions can be tested in parallel. Another, related opportunity is running parallel production and development systems.

There are, of course, some disadvantages. Time-dependent code is illegal on a virtual machine, and channel command words can no longer be dynamically modified simply because the virtual operating system no longer controls real peripherals. However, because time-dependent code and dynamically modified CCWs are no longer considered acceptable programming practice, these restrictions generally apply only to older programs. More relevant is VM’s addition of one more level of overhead, which introduces potential inefficiencies. On a small machine, these inefficiencies could be fatal, but VM is intended for large, fast mainframes.

Traditional operating systems insulate application routines from the hardware. VM’s real potential is derived from the way the control program insulates the user’s entire operating environment from the hardware. Consequently, future hardware changes can be implemented without affecting the existing customer base. As long as the control program acts as an interface between the user’s operating system and the hardware, even radical changes in architecture will remain transparent.

Summary

Traditional multiprogramming operating systems emphasize batch processing, and thus are not effective for program development. Time-sharing supports interactive program development, but most time-sharing systems limit the programmer to a small subset of the com-
puter’s facilities. An ideal program development environment would offer interactive access to the full power of a mainframe. The virtual machine concept is a solution.

On a virtual machine system, a mainframe’s resources are managed by a real operating system that simulates one or more virtual machines. Each virtual machine is the functional equivalent of real computer, with its own virtual memory, virtual operating system, and virtual peripheral devices; in effect, the real operating system multiprograms at the operating system level. UNIX pseudo-computers are one example. This chapter focused on IBM’s VM/SP.

Moving from the second to the third generation was a conversion nightmare. Because the new machines were so different, most programs had to be rewritten, and that was expensive. Today, the investment in software is thousands of times what it was in the 1960s, and new technology that does not maintain compatibility with existing software is probably doomed to failure. Again, the virtual machine concept offers a solution.

Under VM, a single control program (CP) manages the real computer’s resources. Individual virtual machines run under CP. Because CP simulates a real computer for each virtual machine, a variety of virtual operating systems can be supported, including both DOS and OS derivatives. Multiprogramming at the operating system level gives VM tremendous leverage, and allows a computer to support an impressive number of concurrent application routines. One key VM operating system, CMS, simulates a personal computer and thus supports an efficient program development environment.

The control program relies on time-slicing to manage the real processor’s time. A virtual machine is assigned a slice of time. The virtual operating system manages that time. When a time slice expires, CP assigns the processor to a different virtual machine. Generally, interactive virtual machines are given frequent, brief time slices, and noninteractive virtual machines get less frequent but longer time slices.

Segmentation and paging techniques are used to manage memory space. Paging initiated by CP is transparent to a virtual machine. CP treats paging requests initiated by a virtual operating system as I/O operations. The control program maintains page frame tables, paging tables, and segmentation tables for each virtual machine. It is possible to visualize a page moving from an external device, to a virtual machine’s virtual storage, to the real machine’s virtual storage, and finally into real memory.

CP manages all real peripheral devices. Each virtual machine has a set of virtual peripherals. Most nonsharable devices are simulated by spooling. A few, such as the console or a tape drive, are dedicated to a virtual machine. Disk is sharable. When CP generates a virtual machine environment, it assigns disk space to the virtual machine. To CP, each minidisk is a file; to the virtual machine, a minidisk resembles a dedicated drive.

CP maintains a system directory that holds one entry for each virtual machine. A user logs onto CP, which generates the virtual machine environment. The user can then IPL the appropriate operating system and start working.

Interrupts are sensed by hardware, which responds by switching PSWs. Because CP controls real memory, its new PSWs are stored in the real computer’s fixed locations. Thus, following any interrupt, a control program interrupt handler routine is activated.
Normally, CP software switches PSWs, thus simulating the interrupt to the virtual machine. The virtual operating system then handles the interrupt.

Virtual operating systems run in CP’s problem mode. When a virtual operating system issues a privileged instruction, a program interrupt is recognized. If the virtual machine had been in supervisory mode when the privileged instruction was issued, CP performs the requested operation and simulates the results for the virtual machine. If the virtual machine was in problem mode, the interrupt is passed to the virtual operating system and treated as a program exception.

VM supports an effective program development environment. It allows users to move up to more powerful hardware while protecting their existing software investments. Although it does add another level of overhead, its impact is generally positive.

### Key Words

- CMS
- control program (CP)
- conversational monitor system (CMS)
- minidisk
- real computer
- system directory
- transparent
  
- virtual machine
- virtual operating system
- virtual peripheral device
- VM/SP
Exercises

1. Briefly describe the ideal program development environment. Why are batch processing systems, time-sharing systems, and personal computers less than ideal?

2. Distinguish between virtual and real.

3. Briefly explain the virtual machine concept.

4. Real machine details are transparent to the virtual machine. Explain what this means.

5. The fact that two different operating system families, DOS and OS, support applications on IBM mainframes created a serious conversion problem for the company. Explain. How does VM help solve the problem?

6. The virtual machine concept generates considerable leverage, significantly increasing the potential number of concurrent application routines. How?

7. Sketch the components of a virtual machine.

8. Sketch the contents of real memory on a computer running under VM.

9. On the real machine and each virtual machine, the contents of certain fixed memory locations are considered important. Each set of fixed memory locations begins at address 0. How is this possible? Hardware deals with only one set. Which one? Why?

10. The real control program simulates interrupts for the virtual machines. Explain.

11. What is CMS? Why is CMS considered an important component of IBM’s VM/SP?

12. Briefly explain how the control program manages the real processor’s time.

13. Explain how memory is managed under VM.

14. Briefly explain paging under VM.

15. Distinguish between a real peripheral and a virtual peripheral. How does VM handle nonsharable devices? Explain the minidisk concept.

16. Explain how the control program keeps track of its various virtual machine environments.

17. Explain how interrupts are handled under VM.

18. Explain how the control program traps privileged instructions. Why is this necessary?

19. Why is the virtual machine concept so useful?