stroyed. The implementation of this feature is described shortly.

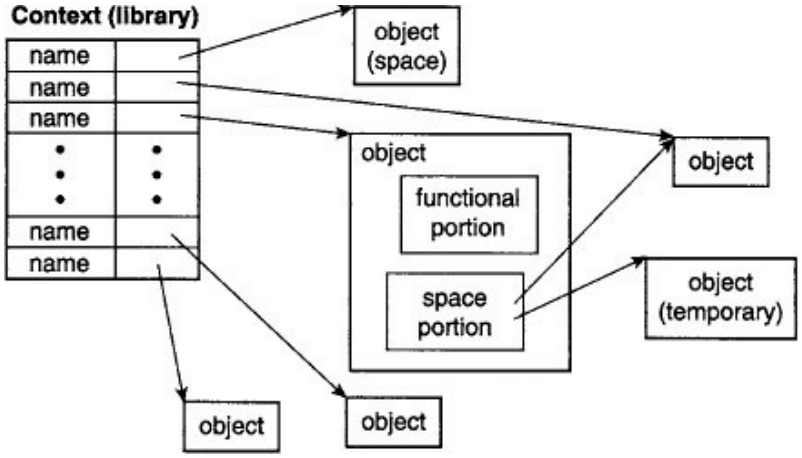



**Figure 7.21** MI Memory Architecture. *Memory consists of objects, which may be either permanent (persistent) or temporary. A typical object contains a functional portion, accessible only via instructions specific to that type of object, and a space portion that can be modified with conventional instructions. The simplest object consists only of a space.*

Protecting the integrity of pointers is an essential part of any object-based system. It is interesting to contrast this method of pointer protection with the one used in the Java VM, where references are protected by carefully tracking their type so that illegal operations on references simply never occur in a properly verified program.

Although the MI object-based memory architecture is reminiscent of the heap used in Java and MSIL, a fundamental difference is that the AS/400 implements a system VM, so objects must persist over the lifetime of the system rather than just the lifetime of a process. There are no conventional files, from the perspective of the OS and applications. All storage, both in DRAM and disk, is one large object-based memory managed beneath the MI level. All long-term storage is done via permanent objects. Each of these permanent objects has a name that is contained in an MI object called a *context* (or a *library* in the operating system, OS/400). To get a pointer to an object, a program can specify a context and an object name; if the program is authorized to access the object, it is then given the correct pointer. This is similar to opening a file in a conventional system. Permanent objects persist in the system until they are explicitly destroyed.

In addition to permanent objects, a process can create temporary objects for its use as it runs. These temporary objects do not necessarily have a pointer held in a context, and they remain in the system until the system is rebooted. This ensures that there are no active programs that have a pointer to a temporary object at the time it is removed. There is no garbage collection in the AS/400, so this is the mechanism by which temporary space is returned to the system.

The implementation of the object-based memory, including all of memory management, is done entirely below the MI level. Main memory is based on the conventional PowerPC segment/page-mapping structure, as described in Appendix Section A.8.1. The object pointers are encoded in 128 bits, but the upper 64 bits contain type information and in some cases authorization information, and the lower 64 bits are essentially a 64-bit PowerPC virtual address.

The only significant extension to the PowerPC memory architecture is the adding of protection for object pointers, i.e., to prevent a program from overwriting a pointer in memory with an arbitrary value, and then later using it for an unauthorized memory access. This is accomplished by adding special instructions for loading and storing pointers and adding a “65th” bit to each memory doubleword. This 65th bit indicates whether the location contains a pointer (actually half of a 128-bit pointer). A load-pointer instruction checks this pointer bit; if set, it sets a condition flag that can be checked to verify that a valid pointer was loaded. A store-pointer instruction stores a pointer and sets the pointer bit(s). Every conventional (nonpointer) store always clears the pointer bit. Hence, any attempt to overwrite a pointer with a regular store results in what will be recognized as a nonpointer if it is ever accessed via a load-pointer instruction.

## 7.8.2 Instruction Set

The MI instructions are not intended to be executed directly, nor are they intended to be interpreted (although, in theory, they could be). They are essentially in a form that must be further compiled (translated) to the host platform's ISA before they can be directly executed. In the original System/38 this was the IMPI; in the more recent systems, it is an extended PowerPC ISA. The MI ISA performs fairly conventional operations that can operate on normal data types held in a space. It also has instructions that operate on MI-defined objects.

The MI instruction format, shown in **Figure 7.22**a, consists of an opcode and operand fields. The opcode is two bytes; the opcode extender (described shortly) is also two bytes. There are zero to *n* operand fields, each three bytes long (in the original System/38, two bytes). For basic arithmetic and logical instructions there is a long format and a short format. The long format is *operand1 ← operand2 op operand3,* where "op" is the operation being performed. The short form is *operand1 ← operand1 op operand2,* i.e., the first operand is both a source and a destination.

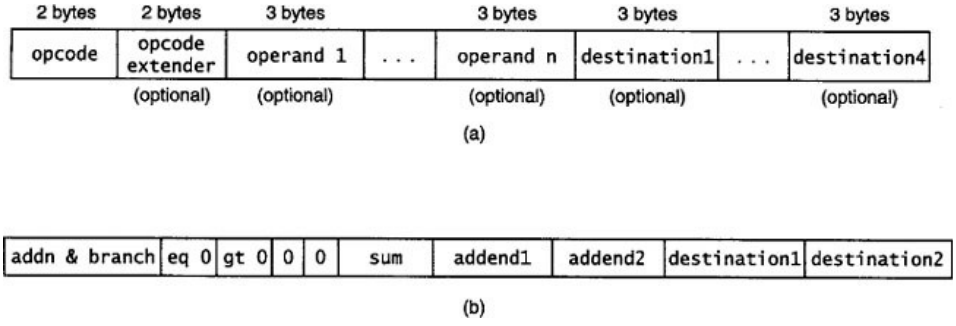



**Figure 7.22** AS/400 Instructions. *(a) The general instruction format; (b) an example instruction that performs an addition and a multiway conditional branch.*

An interesting feature of the MI is that every instruction can include a four-way conditional branch or can evaluate up to four predicate values (called *indicators* in AS/400). These are implemented via the extended opcodes. If there is an opcode extender, then the instruction also has up to four branch destinations (or addresses for indicator values). The 16-bit extended opcode is divided into four 4-bit

fields. Each of the 4-bit fields encodes a condition, such as *greater than zero, equal to zero, less than or equal to zero,* etc. If the opcode extension is for a branch and the result of the instruction matches the first test condition, then there is a branch to the first branch destination; if it matches the second condition, it branches to the second destination, etc. A condition of zero indicates no test (and no destination is given). Similarly, for an indicator-setting instruction, the four destination fields correspond to up to four locations in memory where true/false predicate flags are stored.

For example, the instruction in **Figure 7.22**b does the following—it performs an addition on *addend1* and *addend2,* and the result is placed in *sum.* Then if the sum is zero, it branches to *destination1;* if the sum is greater than zero, it branches to *destination2;* otherwise, there is no branch, and the next instruction is executed. This instruction takes a total of 19 bytes, but it is important to keep in mind that this is the architected *representation* for the instruction. Before it is actually executed, it is translated to an implementation-dependent form that will be more compact, and it does the work of multiple RISC instructions. This instruction, for example, would likely compile to three PowerPC instructions, an add followed by two conditional branches.

The next matter is the way the operands are specified. The operand fields point into a two-level table that contains descriptors of the operands; this is called the *object definition table,* or ODT. As used here, the term *object* is being overloaded; these so-called objects are different from the true architected objects to be discussed later. The ODT, along with an instruction referencing it, is illustrated in **Figure 7.23**. The first-level table is the ODT direction vector (or ODV). This table consists of up to 16 million fixed-length fields that are directly addressed by the 3-byte instruction operand fields. The entries in the ODV contain a description of an operand type along with an optional pointer to the second-level table, the ODT entry string (OES).

The OES is used for those operand descriptors that don't fit in the fixed-length ODV entry.

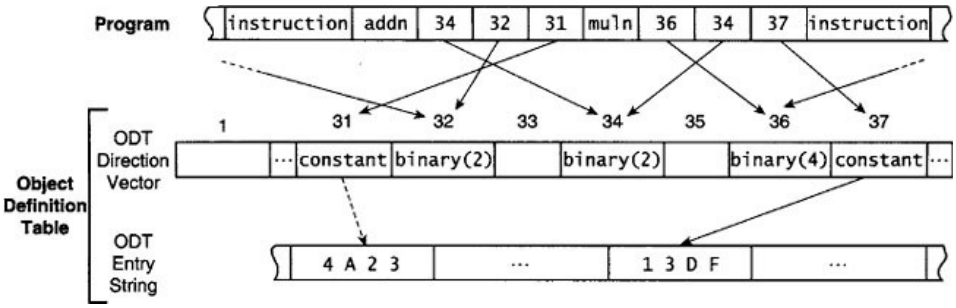



**Figure 7.23** Specifying AS/400 Operands. *AS/400 operands are specified by an object definition table (ODT).*

In the example shown in **Figure 7.23**, two instructions in a program are shown along the top. The first instruction is an addn (add numeric). Its first operand is specified in ODT entry 32 and the other is in ODT entry 31. These operands are 2-byte binary (two's complement) numbers. The second operand is a constant (x4A23), and the pointer in ODT entry 31 points to the OES entry that holds the constant value. The result is also a 2-byte number (as indicated by entry 34). The second instruction takes the result of the first as one operand and a second constant (x13DF) as the second and multiplies them to form a 4-byte binary result. In these arithmetic instructions the opcodes, add numeric and multiply numeric, are generic; the actual type of operation (two's complement, decimal, or floating point) is determined from the types of the operands given in the ODT.

Note that the entries in the ODT indicate the types of operands and the data flow (e.g., that the result of the addn is an input to the mu1n). The actual storage locations for operands are assigned only after the MI is translated to the implementation instructions. At that time the specified operands may be assigned to registers, to a VMM-managed region of memory, or both. The constant operands would likely be translated into immediate fields of implementation instructions. This is very much like the register assignment process ordinarily performed in a compiler backend; here, it is deferred to the

VMM translator.

As pointed out earlier, all logical memory resides within objects; therefore accessing operands in memory must be done through pointers. Pointers are another operand type that can appear in the ODT. A pointer is returned when an object is created or when a symbolic pointer (e.g., as would be held in a context) is resolved.

As noted earlier, some instructions are object specific. The list of MI-supported objects is given below. Note that because the AS/400 was targeted at commercial applications, it has a number of objects intended to support databases. The operating system can use these objects directly for its purposes (the OS is also object based), or it can use these objects as primitives to build OS-defined objects.

***Access group, Context***—objects used for object management

***Authorization list, User profile***—objects used for supporting security

***Byte string space, Data space, Dump space, Space***—objects whose primary use is to hold data

***Commit block, Cursor, Data space index, Dictionary, Index, Journal port, Journal space***—objects whose primary use is to support database operations

***Class-of-service description, Mode descriptor, Process control space, Queue, User profile***—objects for process management, used by the OS

***Controller description, Logical unit descriptor, Network description***—objects used to support I/O

***Module, Program***—objects that contain translated code

## 7.8.3 Input/Output

Because of the focus on commercial applications, the I/O system in the AS/400 is a very elaborate one, based on I/O processors (IOPs), separate from the central processor. Consistent with the overall AS/400 philosophy, I/O is divided into implementation-independent and implementation-dependent parts, with the MI separating the two. The presence of IOPs simplifies the task of pushing the device-dependent aspects out of the central processor (and well below the MI layer).

At the level of the MI there is no secondary (disk) storage; rather, it is part of the unified memory architecture. In other words, disks are not considered to be part of the I/O system, and they have no presence above the MI level. All disk management software, drivers, etc. exist in the implementation-dependent part of the system. It should be pointed out that taking this approach is simplified in an environment where a single company is designing the entire system, i.e., where there is a high degree of vertical integration. The I/O devices visible above the MI layer are divided into generic types, for example, printers, keyboards, graphics displays. The logical characteristics of each of these general types are characterized by MI-level system objects called *logical unit descriptors.* Similarly, device controllers that may be present in the system can each control multiple devices. Two other MI-level objects, the controller descriptor object and network descriptor object, contain the logical characteristics of device controllers and network interfaces. The operating system interacts with software below the MI level (and, in effect, with the I/O devices) through instructions that operate on the MI-level objects.

## 7.8.4 The Processor Resource

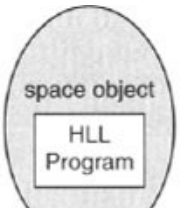
Above the MI level, the operating system establishes overall usage policies, for example, what the priorities the processes should be and what quotas of various resources should be. The actual imple-

and what quotas of various resources should be. The actual implementation of the policy is below the MI layer. That is, the implementation-dependent part of the system actually manages the scheduling queues.

Overall, many conventional operating system functions are done below the MI. This again reflects the overall integrated structure of the AS/400 design environment. When both the platform designers and the OS designers work at the same company (and largely at the same company site), it is easier to divide the OS functions in this manner.

## 7.8.5 Code Translation and Concealment

One of the more interesting aspects of the AS/400 is the way that it manages programs while concealing the implementation-dependent details. The process of compiling and translating a program is illustrated in **Figure 7.24**. First, a high-level language program is compiled, with the resulting MI code and object descriptor table being placed into a space object. Here, it is referred to as a *template* for an object yet to be created. Then a create program instruction is executed. One of its source operands is a pointer to the space object holding the program template. This instruction is "executed," which in fact results in the complete translation of the template contents into a program object containing the implementation-dependent executable file, e.g., PowerPC code. The create program instruction returns a result operand, which is a pointer to the program object containing the translated code. After being created, a program can be executed, and it can be maintained persistently or temporarily, just as any other object.



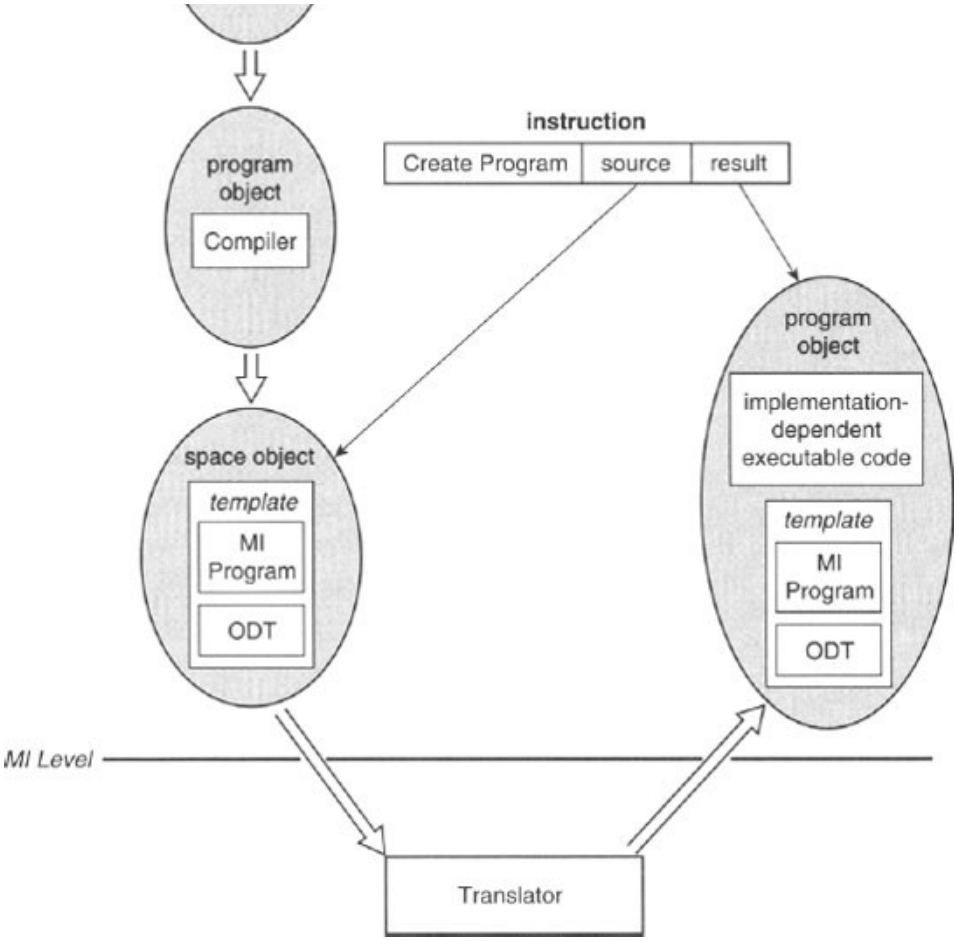



**Figure 7.24** Compiling and Translating a Program on the AS/400. *The steps of compiling an HLL program and translating it into an implementation-dependent form.*

The contents of the program object cannot be directly observed above the MI level; i.e., the contents are concealed, just as with any other object. However, there are times when it may be desirable to look at the program code, for example, when debugging. To accomplish this, the original program template is also kept as part of the program object. Then the original program can be *materialized,* that is, given back to the user in its original, machine-independent form.

This feature yields another interesting capability. Say that a program has been compiled on a given hardware platform and is stored as a persistent object. Then the hardware platform is changed, as when IBM changed the AS/400 platform to PowerPC. Because the original template is stored in the program object, the platform switch is transparent to the user; in particular, the user

does not have to recreate the program object for the new host platform. Rather, when the user attempts to execute the program object on the new platform, the software below the MI layer will recognize that the translated code is for a different platform and use the original program template to produce a translation for the new host platform and place it in the program object.

Finally, it is interesting to compare this overall approach to implementation-dependent concealment with that used by the other codesigned VMs discussed in this chapter. The other codesigned VMs perform both translation and caching of translated code in a concealed manner. This means that each time a program is executed, it must be translated from scratch and then held in a code cache until the program terminates, at which time the code cache contents are discarded. In the AS/400, the fact that programs must be translated is not concealed, yet the actual translated code is concealed (within an object). Because the translation process is not concealed, the system can maintain translated programs in a persistent way, by treating them like any other persistent objects.

# 7.9 Summary

To some extent, modern codesigned VMs, as exemplified by the Transmeta Crusoe, should still be considered a somewhat experimental approach to computer design. This approach is based heavily on dynamic translation to a concealed code cache. Translations do not persist between successive program runs, so translation overhead is a major concern. On the other hand, it takes tens of milliseconds to access the disk when loading a conventional program or when performing other loading and linking tasks, possibly including additional dynamic loading and linking, as the program runs. When viewed from this perspective, the time required for dynamic translation does not appear so great, and, in a sense, it could be considered part of the loading process. Furthermore, the shift to

be considered part of the loading process. Furthermore, the shift to just-in-time compilation for HLL VMs will tend to make this approach appear more conventional as time passes.

The earlier AS/400 object-based approach, although successful and still used in the IBM iSeries machines, has not been copied in other commercial products. It is based on an unconventional separation of hardware and software functionality that probably works best in a completely vertically integrated system development environment. This design environment is much less common today than it was when the AS/400 (System/38) was first developed. A recent research project at the University of Illinois (**Adve et al. 2003**) advocates a new V-ISA that is in some respects similar to the AS/400 MI; it uses object-based memory architecture and a register-intensive instruction set. A primary goal is passing performance-critical information down to the hardware, where it can be more readily used. In other related work, the DELI project (Desoli 2002), builds on the earlier HP Dynamo project, and provides a callable interface into the emulation layer, thereby allowing application or system software the ability to interact with the emulation process.

Thus far the only real example of a modern codesigned VM is the Transmeta Crusoe, although the IBM Daisy/BOA project was carried far into software development and simulation. Both of these systems use a target ISA based on VLIW, and the suitability of VLIW for general-purpose computing remains a topic for debate. The Transmeta processors are targeted at power efficiency; their performance is not at the same level as the high-end superscalar processors that dynamically reorder instructions. Because most contemporary processors use out-of-order superscalar designs, when evaluating performance it is difficult to separate the codesigned aspect from the VLIW aspect to reach any conclusions regarding the overall suitability of the codesigned VM approach. The bottom line is that the codesigned VM approach remains a very interesting technology whose full potential remains to be fully determined.

Finally, an interesting future application of codesigned VMs is to target multiple source ISAs with the same platform. That is, to develop a single processor capable of executing software from multiple, different ISAs, depending on the emulation software that has been loaded into concealed memory. These *convergence architectures* (**Gschwind et al. 2000**) would, among other things, allow computers in server farms to be dynamically customized to fit the software they are called upon to execute.

---

1. Calling the codesigned software *micro-code* indicates that it was considered to be part of the hardware design, as opposed to being considered conventional software.

Previous chapter
< Chapter 6: High-Level Language Virtual Machine Implementation

Next chapter
Chapter 8: System Virtual Machines >

# Chapter Eight

# System Virtual Machines

It was observed very early that many of the hardware resources in a typical computer system are underutilized. The concept of *time sharing* was developed to improve resource utilization by allowing multiple users to access a single computer system simultaneously, with each user being given the illusion of having access to a full set of system resources. In order to create this illusion, a multiprogramming operating system essentially implements a process virtual machine for each application-level program and switches resources among the programs on a time-shared basis.

System virtual machines take this concept one step further by providing a similar illusion for complete systems. A system VM environment is capable of supporting multiple system images simultaneously, each running its own operating system and associated application programs. Each operating system controls and manages a set of virtualized hardware resources, as illustrated in **Figure 8.1**. The virtual resources include a processor (or processors), storage resources, and peripheral devices needed to perform input and output (I/O) for the system.

In a system VM environment, real resources of the *host* platform are shared among the *guest* system VMs, with a layer of software, the virtual machine monitor (VMM), managing the allocation of, and access to, the hardware resources of the host platform. The VMM owns the real system resources and makes them available to one or more guest operating systems that alternately execute on the same hardware. Thus, a guest operating system in a virtual machine environment is given the illusion of owning the resources it subsequently allocates to its various user programs

allocates to its various user programs.

Each virtual resource may or may not have a corresponding physical resource. When a corresponding physical resource is available, the VMM decides how access will be provided to virtual machines requiring its use; e.g., the resource may be partitioned or may be time shared. When a virtual resource does not have a matching physical resource, the VMM may emulate the action of the desired resource, typically via a combination of software and other resources that are physically available on the host platform.

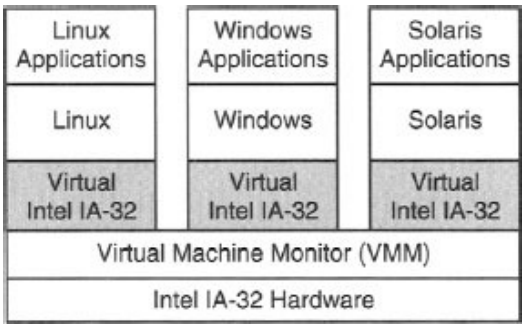



**Figure 8.1** Example System VM Environment. *In this example, the Intel IA-32 platform is capable of simultaneously running the Linux, Windows, and Solaris operating systems, along with their applications.*

System virtual machines have a number of useful applications, some historical, some relevant today, and many that are likely to be important in the future. Some of these applications are described next.

- **Implementing multiprogramming:** Using multiple single-user virtual machines is a simple way of providing multiprogramming and time sharing without requiring a complete multiprogramming OS. Each user in a multiuser environment is given the illusion of having an entire machine, including the CPU, memory, and peripherals. Many of the users of VMs on early mainframe computers, such as the IBM System/370 and its successors, used virtualization as an alternative to a time-sharing system. These users ran a simple, compact, and efficient single-user operating system called the *conversational monitor system* (CMS), instead of the monolithic time-sharing operating systems available at the time.
- **Multiple single-application virtual machines:** This is an extension of the multiple single-user virtual machine concept. Running each application in its

multiple single-user virtual machine concept. Running each application in its own virtual machine increases the robustness of the system. Erratic behavior of one application and its associated OS on a conventional system could bring down the machine. If this happens on a virtual machine, it is less likely to affect the operation of another application running on a different virtual machine. This may be useful in situations where a new application might corrupt the entire system because of either bugs or viruses.

- **Multiple secure environments:** A system VM provides a *sandbox* that isolates one system environment from other environments, thus ensuring a level of security that may not be provided by a single operating system. For example, a user accustomed to running applications on a private machine may be reluctant to transfer the applications to a Web server unless there is some guarantee that the user's resources and activity cannot be accessed or monitored by other users of the same server. In such a situation, a virtual machine can be used to provide an environment that, for all practical purposes, is isolated from the others and through which it is not possible for one user to observe or change another's data and activity.
- **Managed application environments:** In an organization that provides a core set of supported applications to users who also want to install their own applications, VM technology allows the core set of applications to be placed on one virtual machine, protected from other virtual machines on which users install new applications or develop their own programs.
- **Mixed-OS environments:** A single hardware platform can support two different operating systems concurrently. A user may prefer office productivity tools that run on one operating system, for example, but prefer another operating system for application development. The user can do both by installing two virtual machines on a single hardware platform, one for each operating system.
- **Legacy applications:** When a new version of an OS is released, it is not uncommon for OS developers to support seldom-used features with degraded performance while supporting new features that allow most new applications to run with improved performance. In this situation, users who need to run old applications without degraded performance can use one virtual machine to run the "legacy" application under the old OS while using another VM to take advantage of the presumably improved features in the newer version of the OS.
- **Multiplatform application development:** Software developers often need to implement and support software that works on multiple operating systems. Virtual machine technology allows software developers to test their software using multiple virtual machines on the same hardware platform. This is both more convenient and more cost effective than having multiple sets of hardware, one for each OS.
- **New system transition:** A system virtual machine allows a user to migrate to a

- **New system transition:** A system virtual machine allows a user to migrate to a new operating system in a gradual way. The user can try out a new operating system on one virtual machine even while the old operating system is running applications on a different virtual machine. The old operating system can be removed from the system after it has been ascertained that all relevant applications work correctly on the new operating system.
- **System software development:** In large systems, it is sometimes necessary to develop (or test) new system software while the same system is being used for running important applications. In this situation it could be very costly if a bug in the code under development were to bring down the entire system. This situation can be avoided by encapsulating the development environment on a virtual machine separate from the virtual machine where the production-level applications are running.
- **Operating system training:** It is often necessary to demonstrate techniques and effects of changing parameters or policies when training system management personnel. Such training is better done on a copy of the operating system running on a virtual machine than on the operating system running on the real machine so that other users of the operating system do not suffer from unexpected effects.
- **Help desk support:** Help desk service personnel can bring up a virtual machine that emulates the hardware configuration of a client in order to determine the nature and cause of problems the client may be encountering. It is not necessary to have all possible configurations of hardware physically available—virtual machine technology allows a single hardware platform to be set to any of several different configurations.
- **Operating system instrumentation:** Running the operating system on a virtual machine allows the virtual machine monitor to instrument accesses to hardware resources selectively. Not only can all events of a particular type, e.g., page faults, be counted, they can also be logged with detailed information about their nature, their origin, and how the requests were satisfied. Moreover, the programming for all such instrumentation and measurements can be encapsulated outside the operating system, communicating solely with the virtual machine monitor. Today, it is quite common for operating system researchers to conduct most of their experiments using virtual machines rather than directly on the hardware platform. **Keefe (1968)** describes early work on IBM mainframes on system evaluation using a virtual machine. The User Mode Linux system (UMLinux 2003) also uses virtual machine technology to test the fault tolerance of a Linux system.
- **Event monitoring:** Some virtual machines provide capabilities that cannot be performed on a native system. For example, virtual machines lend themselves better to providing traces of execution or dumps of the machine state at points of interest. Similarly, the ability of virtual machines to replay a system execution

interest. Similarly, the ability of virtual machines to replay a system execution from some saved state is useful for analyzing the behavior of a system to understand unexpected behavior.

- **System encapsulation:** System virtual machines provide a convenient way of encapsulating the state of the entire machine. This is useful for checkpointing machine state, with a view to resuming execution at a different time or even on a different host machine.

We are nearly ready to describe system VMs in more detail. Before doing so, however, we first point out that this chapter considers a very important class of system VMs but not all system VMs. In particular, as described in the taxonomy given in **Section 1.5**, this chapter will focus on system VMs where the ISA of the host and guest are the same. Furthermore, most of the attention will be given to uniprocessor systems. Important aspects of VM implementation related to other types of system VMs can be found elsewhere in this book, as summarized shortly.

Multiprocessor virtualization is a simple extension of uniprocessor virtualization. However, there are some special considerations when a multiprocessor is being virtualized. One consideration is that with a single processor, virtualization must be done by time sharing the single processor resource. With multiprocessors, however, virtualization can be achieved by partitioning the multiple hardware processors among the virtual systems. The choice between time-sharing processors and partitioning (or using both) opens up a number of additional design alternatives when multiprocessors are being virtualized.

Another important consideration for shared-memory multiprocessors, particularly when the host and guest ISAs are different, is the memory models supported by the guest and host platforms. The two aspects of memory models that are crucial are memory coherence and memory consistency. Both of these involve the way in which memory operations performed by one processor are observed by other processors in the systems. If the host system does not support the same coherence and/or consistency models as the guest as-

sumes, then the virtualization process may need to take special steps to ensure compatibility. We discuss considerations that are specific to multiprocessor virtualization in **Chapter 9**.

When the guest and host ISAs are different, software in the VMM can emulate the virtual ISA using the techniques described in **Chapters 2**-**4**. **Chapter 9** will address some of the system VM issues that arise where the ISA of the guest is different from that of the host.

# 8.1 Key Concepts

In this section we provide an introduction to the various parts of a hardware platform, with a view to understanding how virtualization can be achieved.

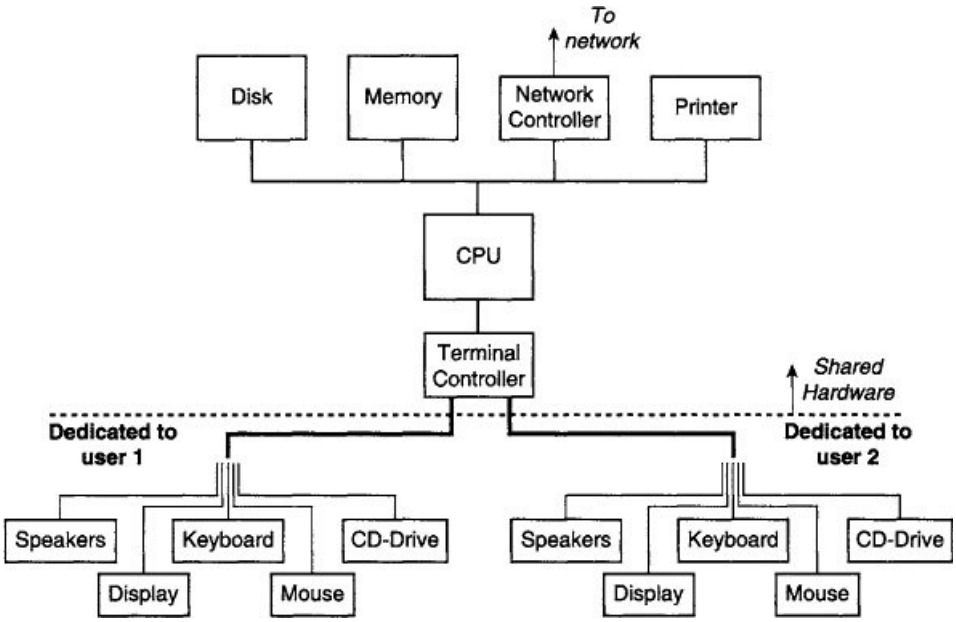



**Figure 8.2** Hardware Replication for a Two-User Virtual Machine System. *The terminal controller collects and tags the requests from multiple users. If such a piece of hardware is not available, the computer backplane must have enough slots to accommodate the adapters for both sets of personal devices.*

## 8.1.1 Outward Appearance

The illusion of multiple machines is an important part of many VM systems. This illusion can be accomplished purely by software means or, in some cases, through a replication of a subset of the hardware resources. For example, there may be cases where the physical resources used directly by individual users—namely, the keyboard, display, and personal peripheral devices, such as a CD ROM drive—are replicated, with one dedicated set for each user, while the rest of the hardware is shared via virtualization software. This situation is shown in **Figure 8.2**.

Alternatively, when a single user wishes to run two operating systems on the same hardware, there is no need to replicate any of the physical devices. Yet the user can be provided with the illusion of running both operating systems simultaneously. By using a hardware switch or, more typically, by entering a special key sequence on the keyboard, all the personal devices can be switched from one virtual machine to the other.

In some system VM environments, one of the operating systems may be considered more important than the other. The user interface of the first provides a window displaying the full user interface of the second. Interactions with applications running on the second operating system take place within this window. In a hosted virtual machine, for example, if the host operating system were Windows, a graphical window could be established on the desktop interface to interact with a virtual machine on the same platform. This solution is similar to the DOS window that may be brought up within the Windows desktop interface to run legacy DOS applications.

## 8.1.2 State Management

The architected state of a computer is contained in and maintained by the hardware resources of the machine. From a performance point of view, all state-holding hardware resources are not equivalent. There is usually an architected hierarchy of state resources,

ranging from registers at the top of the hierarchy to secondary storage (e.g., disks) at the lower end of the hierarchy. In a VM system, each virtual machine has its own architected state information, and there may or may not be adequate resources in a host hardware platform to map each element of a guest's state to its natural level in the host's memory hierarchy. For example, a guest's register state may actually be held in the main memory of the host platform as part of a register context block.

In normal operation, the VMM periodically switches control among the guest VMs. Regardless of where the state is actually held, as operations on the guest's state are performed (e.g., via instruction execution), the state maintained on the host machine should be modified just as it would be on the guest's native platform. This is essentially the VM isomorphism discussed in the book's introduction and illustrated in **Figure 1.2**. There are two basic ways of managing guest state so that this is accomplished.

One way is to employ a level of indirection, by holding the state for each guest in fixed locations in the host's memory hierarchy with a VMM-managed pointer indicating the guest state that is currently active. As the VMM switches among guest systems, it changes the pointer to match the current guest. This is illustrated in **Figure 8.3**a, where the pointer actually points to the register context block of the currently active guest virtual machine. This is analogous to the situation in virtual memory systems, where the page table pointer is used by the operating system to point to the address space of the currently active process.

The approach of using indirection can be relatively inefficient if the memory resource that holds state in the guest VM has characteristics that are different from those on the native platform, as in the example of **Figure 8.3**a, where the register state is actually held in memory. In such a case, in order to perform a copy operation from register A of virtual machine 2 to register B of the same virtual ma-

chine, the VMM would have to perform a memory-to-memory copy by loading the value in the memory location representing register A into some temporary register and then storing the value in the memory location corresponding to register B. Such a copy takes two memory-access instructions and could take more instructions if an address register must be loaded with a pointer to the location of the register file in memory.

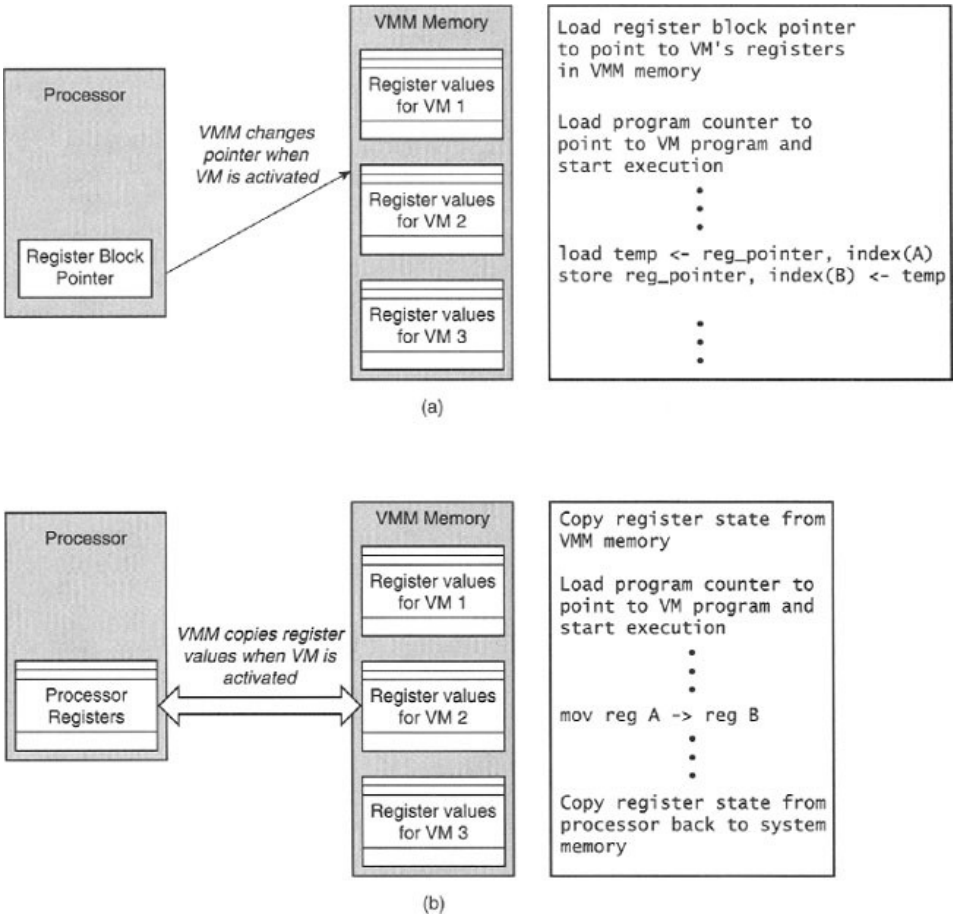



**Figure 8.3** Methods of Maintaining Virtual State Illustrated for a Register File. *(a) Using indirection, (b) by copying. The boxes at the right indicate the actions taken when activating a virtual machine and copying a register value. In the first case, the value corresponding to a register is loaded from a table in memory and copied back to another location in the table. In the second case, the registers are used directly to accomplish the move.*

To avoid this inefficiency, a second approach for managing guest state is to copy a guest's state information to its natural level in the memory hierarchy whenever it is activated by the VMM and to copy it back when a different guest is activated. For example, in **Figure**

8.3b, the VMM copies the entire guest register contents into the host's register file at the time virtual machine 2 is activated by the VMM (after saving the registers of the previous guest). With this approach, the operation of moving a value from register A to register B is achieved by directly executing a native register-move instruction. In many system VM implementations, the goal is to execute guest code natively (i.e., without emulation) on the host platform. Thus, in the register example, after the one-time overhead of switching out the old virtual machine registers and switching in the registers of virtual machine 2, execution can proceed at the same rate as on a machine running the guest natively.

The choice between indirection and copying depends on the frequency of use and whether the guest state being managed by the VMM is held in a different type of hardware resource than on a native platform. For frequently used state information, such as the general-purpose registers, it is usually preferable to swap the state of the virtual machine to the corresponding real resource each time the virtual machine is activated.

### 8.1.3 Resource Control

Within a system VM environment, hardware resources, including the processor resource, are assigned to a VM at the time it is created according to configuration specifications. Once resources are given to a guest VM, it is important that there be a way for the VMM to get them back so that they can be assigned to a different VM. Thus, the VMM must maintain overall control of all the hardware resources, even though they are temporarily being used by the guest VM currently running. This section examines ways in which this can be achieved.

First note that a similar issue arises in the case of conventional time-sharing systems—multiple jobs are concurrently being executed on the machine, with each job having full access to its resources at any

given instant. However, there are some resources in the machine that are not directly accessible to the user application but accessible only to the operating system. One such resource is the interval timer, which, after being loaded with a value by the operating system, counts down clock ticks and triggers an interrupt when the value it holds becomes zero. Before passing control to a user process, the operating system initializes the timer with the maximum period of time it will allow the user process to run. The timer interrupt will then guarantee that control is transferred back to the operating system within the maximum time interval. Thus, the operating system uses the timer to keep control of the processor by ensuring that no user application can run for an indefinite length of time. And, because the OS has control of the processor, it can then control all the other resources. This control issue is similar to the one discussed in connection with runtime control of emulation frameworks in **Chapter 3**.

The situation is similar in a system VM environment. Here, the principal issue is that of time sharing resources among the different virtual machines. We will see in **Section 8.2.1** that a convenient way to provide the VMM with overall control is for the VMM to intercept all accesses to so-called privileged resources, such as the interval timer. These privileged resources, therefore, are not used directly by the individual virtual machines—rather, the VMM emulates the operation of these resources at all times. Instead of allowing the operating system in a virtual machine to field the timer interrupt, for example, the VMM first handles the interrupt itself. The handler for the timer interrupt includes code that saves the state of the current guest virtual machine, determines the next virtual machine that should be activated, and loads the state for the next virtual machine to be run. It accomplishes this, as mentioned earlier, either by changing pointers, by copying state into hardware resources, or by doing both. **Figure 8.4** depicts scheduling actions taken by the VMM.

There may be additional opportunities for the VMM to get control of the system other than through interval timer interrupts. For example, since privileged instructions encountered in the user mode must, as noted earlier, be intercepted and emulated by the VMM, the VMM gains control whenever the guest operating system issues privileged instructions (the mechanism for doing so is described in **Section 8.2**).

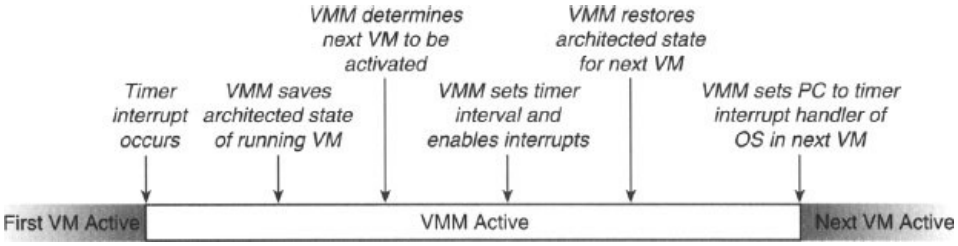



**Figure 8.4** Actions Taken by the VMM in Retiring One Virtual Machine and Activating the Next Virtual Machine.

A fair scheme for allocating hardware resources to virtual machines is one in which the virtual machines get ownership of the resources in turn for roughly equal amounts of time. The issues here are similar to the issues that arise in multiprogrammed operating systems. If the time allocated to each virtual machine is large, there is the possibility that the resources are not optimally utilized. On the other hand, if the time allocated is small, the overhead incurred in switching between the virtual machines begins to degrade performance.

For a fair scheme to work, the guest operating system on a virtual machine must be denied direct access to the interval timer facility so that it cannot reschedule the next timer interrupt. Furthermore, in order to make operation under a virtual machine completely transparent, the guest operating system should not be allowed to read the real timer value set by the VMM. (This transparency is needed to prevent the operating system from behaving in a different manner when running under a VMM than it would on a real machine.) However, as we will see later in the chapter, the VMM can supply the guest with an emulated *virtual* interval timer.

Strict VMM emulation of a guest's timer interval could potentially lead some user code to run for too long a time, however, starving other virtual machines from using the hardware resources. Hence, the VMM must examine every guest setting of the timer interrupt and override the requested value if it has the potential of being unfair to other virtual machines that may be running on the system. This is an instance where the emulation of privileged resources must be done by the VMM with a view to efficient and fair operation of the entire virtual machine system rather than for the performance of an individual guest.

## 8.1.4 Native and Hosted Virtual Machines

It should be clear by now that the VMM is the key component in any system VM environment. The VMM is responsible for scheduling and managing the allocation of hardware resources to the various guest virtual machines. Hence it is also the point of control for the shared physical resources in the system. Such resources include the registers in the CPU, the real memory in the system, and the various I/O devices attached to the system. For *efficient* operation of the system, therefore, at least some part of the VMM implementation should have privileges higher than the actual privileges of the guest virtual machines it supports. By *actual privilege,* we mean the privilege the code has on the hardware while running in the virtual machine environment. This, as we will see shortly, may be different from the privilege that is *perceived* on the guest.

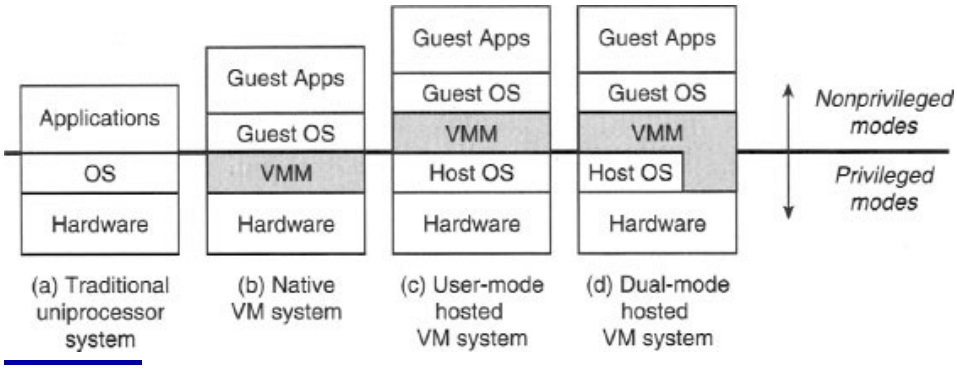


**Figure 8.5** Native and Hosted VM Systems. *(a) The operating system of a traditional uniprocessor executes in the privileged mode. (b) In a native VM system, the VMM operates in the privileged mode. (c-d) A trusted host operating system executes in the privileged mode in a hosted VM system. In both systems, the guest operating system resides in the virtual machine and operates at a less privileged level. The VMM of a hosted system may work entirely in the user mode (c) or in dual mode (d) when parts of the VMM operate in the privileged mode.*

As pointed out earlier, the relationship between the VMM and virtual machines is analogous to the relationship between an operating system and application programs in a conventional time-shared system. In the latter, the operating system typically works in a privilege level higher than that of the applications, e.g., in *system mode* versus *user mode,* as shown in **Figure 8.5**a. A virtual machine system in which the VMM operates in a privilege mode higher than the mode of the guest virtual machines is called a *native VM* system. In a native VM, the VMM is the only software that executes in the highest privilege level defined by the system architecture, as depicted in **Figure 8.5**b. Conceptually, the VMM is first installed on the bare hardware, and the guest operating systems are then installed on top of the VMM. The guest operating systems and other less privileged applications run in levels of privilege lower than that of the VMM. This typically means that the privilege level of the guest OS is emulated by the VMM. Native VM systems have been used and studied extensively and are discussed in detail in upcoming sections.

For user convenience and implementation simplicity, it is often advantageous to install a virtual machine system on a host platform that is already running an existing OS. Such a system is called a *hosted VM* system (in this case the term *host* refers to the underlying OS). In a hosted VM system the VMM utilizes the functions already available on the host OS to control and manage resources desired by each of the virtual machines. Modifying a proprietary commercial operating system is often not possible, either because the source code is unavailable or because of licensing agreements. In these cases, the VMM may be implemented at the user level, a privilege level below that of the host operating system, as shown in **Figure 8.5**c. Such a system is commonly referred to as a *user-mode* hosted

VM system. For efficiency reasons, however, it is desirable to have at least part of the VMM work in the most privileged mode. This can often be achieved by modifying the host operating system through mechanisms commonly provided to extend the functionality of an operating system, such as kernel extensions or device drivers. Such a system that has parts of the VMM operating in a privileged mode and other parts in nonprivileged mode, as depicted in **Figure 8.5**d, is referred to as a *dual-mode* hosted VM system. We will examine features of such systems further in **Section 8.4.3**.

## 8.1.5 IBM VM/370

The first virtual machine environment was the IBM System/360 Model 40 VM (circa 1965) (**Adair et al. 1966**; **Creasy 1981**). The intention of the developers was to build a time-sharing system that extended the then-novel concept of virtual memory to other parts of the computer system. After debating the alternatives, the design team decided that the most appropriate interface to provide to the user would be the full System/360 architecture interface (the ISA) rather than just the user-level architectural interface enhanced by system library functions (the ABI). The perceived advantage was that this would be the best way to protect users from one another while preserving compatibility as the System/360 architecture evolved. It was not until much later, with a model of the System/370, that VM became mainstream. We will refer to the IBM virtual machine in the rest of the chapter as the IBM VM/370—many of the principles are still in use today in the z/VM running on the IBM zSeries mainframes. An informative account of the development of VM on the IBM platforms is provided in **Varian (1997)**, while a survey of virtual machine research at that time appears in **Goldberg (1974)**.

The virtual machine monitor of VM/370 was called the *control program* (CP). The CP design team also developed a single-user operating system, called the conversational monitor system (CMS), mainly

ing system called the *conversational monitor system* (CMS), mainly to demonstrate the advantages of modularization for system evolution. The CP/CMS design separated the function of resource management from the function of providing services to the user. The success of the project was in large measure due to the design of the System/370 architecture, which allowed an elegantly simple implementation of the virtual machine concept.

Even though CP and CMS are often referred to as a pair, one can exist without the other. In fact, CMS was developed on the bare machine before CP existed, and CP has hosted many operating systems in its life, including, most recently, Linux. As its importance and popularity grew, special hardware features were added to the System/370, primarily to reduce virtualization overhead. Some of these enhancements will be discussed in **Section 8.5**.

## 8.2 Resource Virtualization—Processors

The key aspect of virtualizing a processor lies in the execution of the guest instructions, including both system-level and user-level instructions. There are two ways this can be done. The first is through *emulation.* As described in **Chapter 2**, emulation can be performed through either interpretation or binary translation. Emulation involves examining each guest instruction in turn, either repeatedly when interpreting or once when performing binary translation, and emulating on virtualized resources the exact actions that would have been performed on real resources. Emulation is the only processor virtualization mechanism available when the ISA of the guest is different from the ISA of the host. As we shall see presently, however, an emulation-like process may be necessary at times even when the two ISAs are identical. Such a situation occurs when instructions that interact with hardware resources need to operate differently on a virtualized processor than on a real processor

differently on a virtualized processor than on a real processor.

The second processor virtualization method uses *direct native execution* on the host machine. This method is possible only if the ISA of the host is identical to the ISA of the guest, and even then only under certain conditions. It is always possible to build a virtual machine by using emulation. However, even with sophisticated techniques such as binary translation, the performance of a program on the virtual machine will rarely be as good as its performance on native hardware. Therefore, a basic design goal for system VMs with identical guest and host ISAs is that a significant fraction of the instructions execute directly on the native hardware. If this is possible, a program will often run on a virtual machine at about the same speed as on native hardware, unless there are memory or I/O resource limitations. The overhead of emulating any remaining instructions depends on several factors, including the actual number of instructions that must be emulated, the complexity of discovering these instructions, and the data structures and algorithms used for emulation.

The following three subsections characterize those instructions in an ISA that must be emulated in order for a VMM to be successfully constructed. The techniques used for emulating the special instructions depend on the characteristics of the ISA. We will see that for a well-behaved, efficiently virtualizable ISA, a trap occurs naturally when an instruction needs to be emulated, the trap handler jumps to an appropriate interpreter routine, interprets the single instruction, and returns control back to the original program. In other, less well-behaved ISAs, the difficulty of isolating the instructions that need to be emulated may force a larger number of other instructions to be emulated. **Section 8.2.4** addresses the problem of discovering such instructions. In these cases, emulation performance can be improved by translating a group consisting of several instructions at a time using techniques similar to those described in **Chapter 2**.

## 8.2.1 Conditions for ISA Virtualizability

In a classic paper (**Popek and Goldberg 1974**) inspired by Goldberg's doctoral dissertation (**Goldberg 1972**), Popek and Goldberg formally derived sufficient conditions under which an ISA can efficiently support virtual machines. We summarize in this subsection the valuable insight regarding the construction and understanding of system VMs provided by this work.

In **Section 8.1.4** we characterized both native and hosted system VMs. There tends to be an inherent inefficiency in purely hosted VMs because, as we will see in **Section 8.4.3**, the VMM needs to rely heavily on services of the host operating system. Therefore, we restrict the discussion here to native system VMs. In a native system VM, the VMM runs in system mode, and all other software runs in user mode. Note that this "other" software includes some software originally intended to run in the system mode, a guest operating system being the prime example. The VMM keeps track of the intended (i.e., virtual) mode of operation of a guest virtual machine but sets the actual native hardware mode to *user* mode whenever executing instructions from the guest virtual machine, regardless of whether it is user or system software.

The original analysis by Popek and Goldberg was for "third-generation" machines, such as the IBM System/370, the Honeywell 6000, and the Digital PDP-10, but it still holds for present-day machines (one could argue that from an ISA perspective, we are still in the third generation!). The assumptions made in the analysis are as follows: (a) The hardware consists of a processor and a uniformly addressable memory, (b) the processor can operate in one of two modes, the system mode or the user mode, (c) some subset of the instruction set is available only in the system mode, and (d) memory addressing is done relative to the contents of a relocation register (i.e., a fixed value is added to a virtual address to arrive at a physical

memory address). The analysis could be extended to more than two modes and to more complex memory architectures, but the fundamental results remain unchanged. Input/Output is not considered, but the approach and analysis can also be extended to I/O in a straightforward manner.

The machine being virtualized is modeled as a 4-tuple, S = ≤ *E, M, P, R*>, where *E* refers to the executable storage, *M* refers to the mode of operation, *P* is the program counter, and *R* is the memory relocation bounds register. The memory of the machine is assumed to be located contiguously in physical memory from the location given in the relocation bounds register *R*. The relocation bounds register, as the name suggests, is a pair indicating the physical location and the size of the virtual memory space. The extension to paged or segmented virtual memory systems is straightforward. A *memory trap* occurs if the address accessed by a program falls outside the bounds indicated by *R*. The trap atomically saves the current state of the machine, represented by *M, P,* and *R* in location 0, and copies the contents of location 0 of another process into *M, P,* and *R*—this essentially has the effect of performing a context switch.

A *privileged* instruction is defined as one that traps if the machine is in user mode and does not trap if the machine is in system mode. It is not sufficient for the instruction's behavior to be different in the two modes, and by definition the privileged instruction *must* trap when executed in the user mode. Note that this strict definition is for the purpose of this particular analysis and may differ from definitions used elsewhere when describing specific ISAs. Here are two examples of privileged instructions.

- **Load PSW** (LPSW, IBM System/370): This instruction loads the processor status word (PSW) with a doubleword from a location in memory if the processor is in system mode. If it is not in system mode, the machine traps. The PSW contains bits that determine, among other things, the state of the CPU. One of these bits, the P bit, specifies whether the CPU is in user or system mode.

Another part of the PSW is the instruction address (or program counter). If this instruction could be executed in user mode, a malicious user program could put itself in system mode and get control of the system.

- **Set CPU Timer** (SPT, IBM System/370): This instruction replaces the CPU interval timer with the contents of a location in memory if the CPU is in system mode and traps if it is not. Once again, if this instruction could be executed in user mode, it would be possible for a user program to change the amount of time allocated to it before being swapped out.

As noted earlier, in a virtual machine environment, an operating system running on a guest virtual machine should not be allowed to change hardware resources in a way that affects the other virtual machines or the performance of programs outside the realm of its own virtual machine. Hence, even the operating system on a virtual machine must execute in a mode that disallows the direct modification of system resources such as the PSW and the CPU interval timer. Consequently, all of the guest operating system software is forced to execute in user mode.

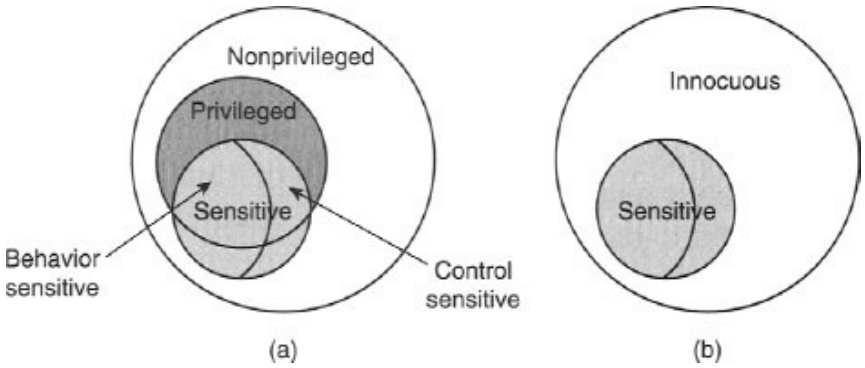



**Figure 8.6** Types of Instructions. *(a) Sensitive and privileged instructions overlap (although not necessarily completely). (b) Sensitive instructions and innocuous instructions are complements of each other.*

To specify instructions that interact with hardware, two categories of special instructions are defined. *Control-sensitive* instructions are those that attempt to change the configuration of resources in the system, for example, the physical memory assigned to a program or the mode of the system. (If I/O were included in the model, then I/O would also be considered a resource.) *Behavior-sensitive* instructions are those whose behavior or results produced depend on the configuration of resources—in the model, this includes the value in

the relocation bounds register or on the mode of operation. If an instruction is neither control sensitive nor behavior sensitive, it is termed *innocuous* (see **Figure 8.6**).

The LPSW and SPT instructions mentioned earlier are examples of control-sensitive instructions. Their use allows an operating system to change some basic resource of the system, such as the mode of operation or the CPU timer. Examples of behavior-sensitive instructions follow.

- **Load Real Address** (LRA, System/370): This instruction takes a virtual address, translates it, and saves the corresponding real address in a specified general-purpose register. The behavior of this instruction (i.e., the result value stored in the register) depends on the state (mapping) of the real memory resource.
- **Pop Stack into Flags Register** (POPF, Intel IA-32): This instruction pops the flag registers from a stack held in memory. One of the flag registers is the interrupt-enable flag, which can be modified only in privileged mode. In user mode, this instruction overwrites all flags except the interrupt-enable flag. For the interrupt-enable flag, the instruction acts as a no-op when executed in user mode. The behavior of this instruction therefore depends on the mode of operation.

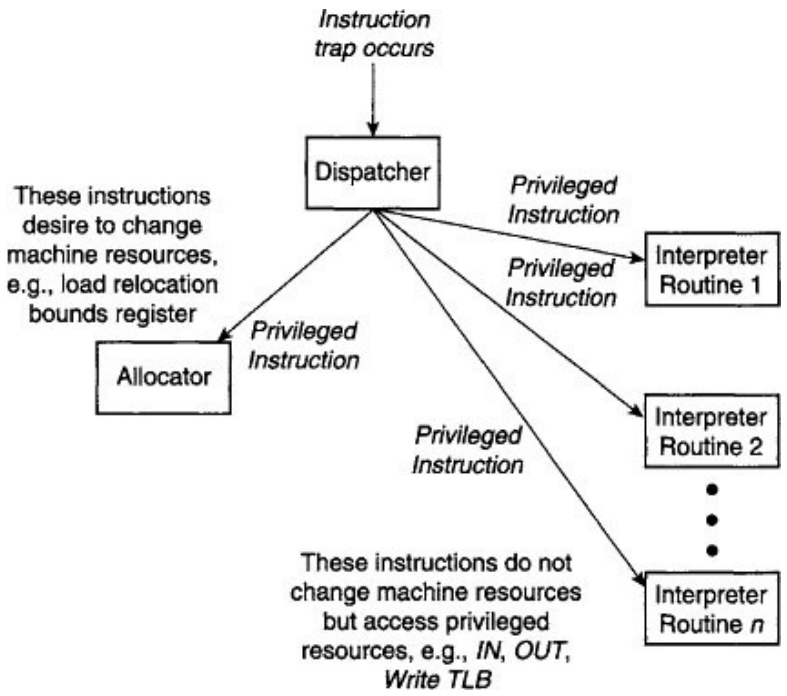


**Figure 8.7** Components of a Virtual Machine Monitor.

The functions of the VMM can be divided into three parts, a *dispatcher*, an *allocator*, and a set of *interpreter routines*. These components of the VMM are depicted in **Figure 8.7**. The dispatcher is the top-level control module of the VMM, which decides the next module to be invoked; it is invoked by the interrupt handler when the hardware traps. The allocator decides how system resources are to be allocated, e.g., how to allocate memory resources in a nonconflicting manner. The allocator is invoked by the dispatcher whenever there is a need to change machine resources associated with some virtual machine.

As we shall see, in a well-constructed VMM, any instructions that attempt to change resource assignments or whose behavior is affected by the assignment of resources will trap to the VMM dispatcher. The trapping instructions that attempt to change resource assignments are then directed by the dispatcher to the allocator. The dispatcher directs all remaining traps to the interpreter routines. The interpreter routines, one per privileged instruction, emulate the effects of the instructions when operating on virtual resources. After an interpreter routine finishes, control is passed back to the guest virtual machine at the instruction immediately following the one that caused the trap.

According to Popek and Goldberg, a potential virtual machine monitor must satisfy three properties before qualifying as a true virtual machine monitor: efficiency, resource control, and equivalence. *Efficiency* implies that all instructions that are innocuous must be executed natively on the hardware, with no intervention or emulation by the VMM. *Resource control* implies that it should not be possible for guest software to directly change the configuration of any system resources available to it, e.g., real memory. The allocator must be invoked if the guest software makes any such attempt. *Equivalence* implies that any program executing on a virtual ma-

chine must behave in a manner identical to the way it would have behaved when running directly on the native hardware, with only a few exceptions. The allowed exceptions are that (1) performance can be slower due to the emulation (e.g., interpretation) of certain instructions, (2) there may be a limitation on the extent of resources available, e.g., disk space, because of sharing among virtual machines, and (3) there may be differences in performance due to changed timing relationships, e.g., between the I/O and the processor.

Here, the reader should note that this formulation of a VMM is somewhat at odds with the common definition and the definition used elsewhere in this book. It is intuitively clear that the resource control and equivalence properties should hold for a VMM. Furthermore, it should always be possible to construct a VMM with these properties, however inefficient, by employing emulation techniques whenever necessary. In effect, it is these two properties that we (and others) typically assume when defining a VMM. In the Popek and Goldberg formulation, a VMM must also be “efficient” in order to be considered a true VMM. And it is the efficiency condition that most heavily depends on the condition stated shortly ahead in Theorem 1.

An alternative formulation would define a VMM to have only the properties of resource control and equivalence and then to define an *efficient VMM* to be a VMM that also satisfies the efficiency property. With the exception of the Popek and Goldberg theorems, we use these alternative definitions of VMMs and efficient VMMs in this book. Here, then, is the key theorem regarding (efficient) VMMs.

**Theorem 1:** For any conventional third-generation computer, a virtual machine monitor may be constructed if the set of sensitive instructions for that computer is a subset of the set of privileged instructions.

This is a surprisingly simple theorem, yet it has important implications. It says that an efficient virtual machine implementation can be constructed if instructions that could interfere with the correct or efficient functioning of the VMM (i.e., sensitive instructions) always trap in the user mode. All nonprivileged instructions can be executed natively on the host platform; no emulation is needed. The trapping of sensitive instructions, such as ones that attempt to reallocate system resources or depend on system resource assignments, force control to go back to the VMM. The VMM can then examine the action desired by the virtual machine that issued the sensitive instruction and reformulate the request in the context of the virtual machine system as a whole. **Figure 8.8** illustrates the sufficient condition contained in the theorem.

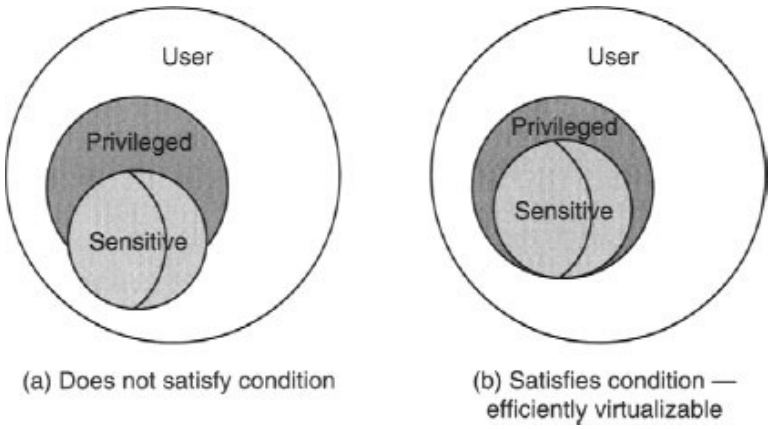



**Figure 8.8** Illustrating Popek and Goldberg's Theorem 1. *In (a), the sensitive instructions are not a subset of the privileged instructions and hence the system may not be efficiently virtualizable. On the other hand, the system in (b) satisfies the condition of Theorem 1.*

The VMM interprets a sensitive instruction according to the prevailing status of the virtual system resources and the state of the virtual machine. Take, for example, the sensitive instructions LPSW and SPT described earlier in this section. These System/370 instructions are also privileged and hence satisfy the condition indicated in Theorem 1. Let's assume that a particular instance of an LPSW instruction has the net effect of going from one privileged state to another state with the same set of privileges. The LPSW interpreter routine in the VMM determines that this particular transfer is harm-

less by examining and comparing the contents of the location being loaded with the contents of the virtual copy of the PSW it maintains for that virtual machine. It then transfers control appropriately by modifying the contents of the hardware PSW, using an LPSW of its own. This LPSW does not trap because the VMM operates in system mode. **Figure 8.9** illustrates this process.

On the other hand, if the LPSW instruction were executed by a user application running on the virtual machine, the trap handler would generate a virtual trap that is then passed to the virtual machine. The guest operating system handles this trap in exactly the same way as a native machine would when an attempt is made to execute a privileged instruction in user mode.

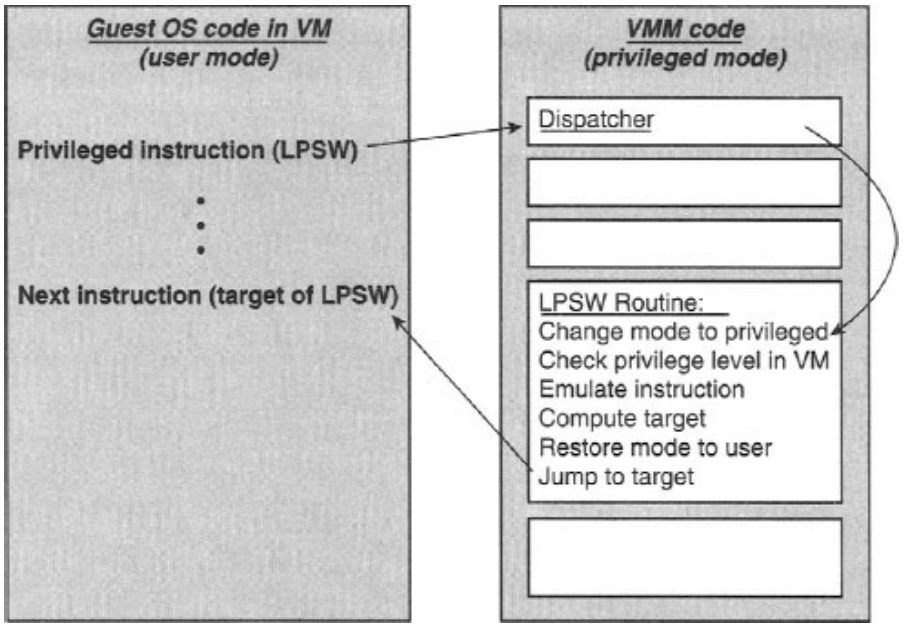



**Figure 8.9** Handling of a Privileged Instruction in a Guest OS. *The privileged instruction causes a trap because the guest OS is operating in user mode. The trap goes to the dispatcher in the VMM, from where it is directed to the special interpreter routine for the specific instruction.*

When interpreting the SPT instruction, the VMM examines the contents of the location to be loaded into the CPU timer. It will allow this value to be loaded if the value (say, $t$) is less than the time remaining from the allocated time for the virtual machine itself ($t \leq T$). Otherwise it will load the timer with the time remaining for the vir-

tual machine (*T*). Meanwhile, it keeps the time difference (*t*—*T*) in an internal table so that this time can be restored when the guest VM is again activated. In this manner, the VMM ensures that it always remains in control, even in the presence of attempts by some virtual machine to monopolize the system resources. If the guest OS should attempt to read the timer value, the VMM will reconstruct the correct virtual timer value and return it to the guest.

There are several simplifying assumptions made concerning the types of architectures under consideration in developing Theorem 1, and some practical aspects such as I/O and asynchronous interrupts have been ignored. Yet the theorem is useful because it suggests a convenient way of testing an ISA to determine the feasibility of an efficient virtual machine monitor. The theorem also provides clear guidelines for computer architects wishing to ensure that a new ISA lends itself to efficient virtualization.

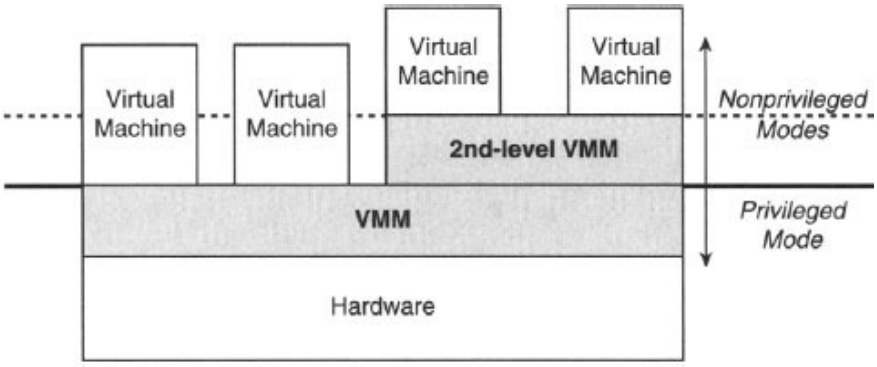



**Figure 8.10** Recursive Virtualization.

## 8.2.2 Recursive Virtualization

It may sometimes be desirable to run a virtual machine system as one of the virtual machines. This implies that the VMM itself would be running in the user mode under the control of a copy of itself running in the privileged mode. The concept of running the virtual machine system on a copy of itself is called *recursive virtualization* (**Figure 8.10**).

There are two effects that usually restrict the ability to create an efficient recursively virtualizable system. First, if the VMM itself has timing dependences, then its performance could be affected adversely when it runs in user mode within another virtual machine. In fact, the presence of timing dependences in a program could end up violating one of the conditions of Popek and Goldberg—the equivalence property. Thus a strong requirement for a recursively virtualizable system is that a VMM without any timing dependences be constructible for the system.

The second effect comes from the fact that each VMM layer uses up its own resources, particularly system memory resources needed to keep the state of the various virtual machines running on the system. If recursive virtualization is repeatedly performed, then these resources keep shrinking to a point where little is left for allocation to the virtual machines. As a result, the number of programs that run with identical effect on the real and virtual machines shrinks as the number of levels of virtualization increases.

Neither of these is a problem, in practice, however. Seldom does the required number of levels of virtualization go beyond two. At this level of recursion, the overhead is still contained; moreover, the user is usually willing to accept the drop in performance resulting from the recursive virtualization.

Popek and Goldberg formalized the conditions for recursive virtualizability in a second theorem.

**Theorem 2:** A conventional third-generation computer is recursively virtualizable if (a) it is virtualizable and (b) a VMM without any timing dependences can be constructed for it.

## 8.2.3 Handling Problem Instructions

Observe that, as described earlier, the POPF instruction in the Intel IA-32 ISA is sensitive but not privileged—it does not generate a trap in user mode. Hence it violates the virtualizability condition of Theorem 1. The Intel IA-32 has several other instructions that similarly are sensitive but not privileged (**Robin and Irvine 2000**). In reality, there are very few ISAs that are virtualizable in the sense of the Popek-Goldberg theorem (or efficiently virtualizable in the terminology we use). However, if an ISA is not efficiently virtualizable, all is not lost. It simply means that an additional set of steps must be taken in order to implement a system virtual machine (with possible loss of some efficiency).

The POPF instruction mentioned earlier is a problem instruction because it inhibits the creation of an (efficient) virtual machine monitor in the sense of Theorem 1. For convenience, let us refer to instructions that are sensitive but not privileged as *critical* instructions. It would be possible for a VMM to intercept POPF and other critical instructions if all guest software were interpreted instruction by instruction. The use of interpretation clearly leads to inefficiency (both as formally defined by Popek and Goldberg and in reality). Fortunately, techniques related to those described in **Chapters 2** and **3** can be used to reduce the inefficiency considerably. For example, the VMM can scan the guest code stream before execution, as done during binary translation, and in the process discover all critical instructions, replacing them with a trap or a jump to the VMM, as shown in **Figure 8.11**. This replacement process is known as *patching* (**Section 4.7.1**). Only the code running on the guest virtual machines needs to be scanned—the VMM code is in privileged state and will work correctly without modification. The problem of scanning and patching critical instructions is discussed in more detail in the next subsection.

In a strictly efficient virtualizable machine, as defined by Popek and Goldberg, all nonprivileged instructions are executed natively. We refer to a virtual machine system in which some of the nonprivi-

refer to a virtual machine system in which some of the nonprivileged instructions must be emulated as a *hybrid virtual machine* system. Scanning and patching is a pragmatic attempt by a hybrid virtual machine system to execute *most* nonprivileged instructions natively. Guest VM software containing few critical instructions should not see a significant degradation in performance.

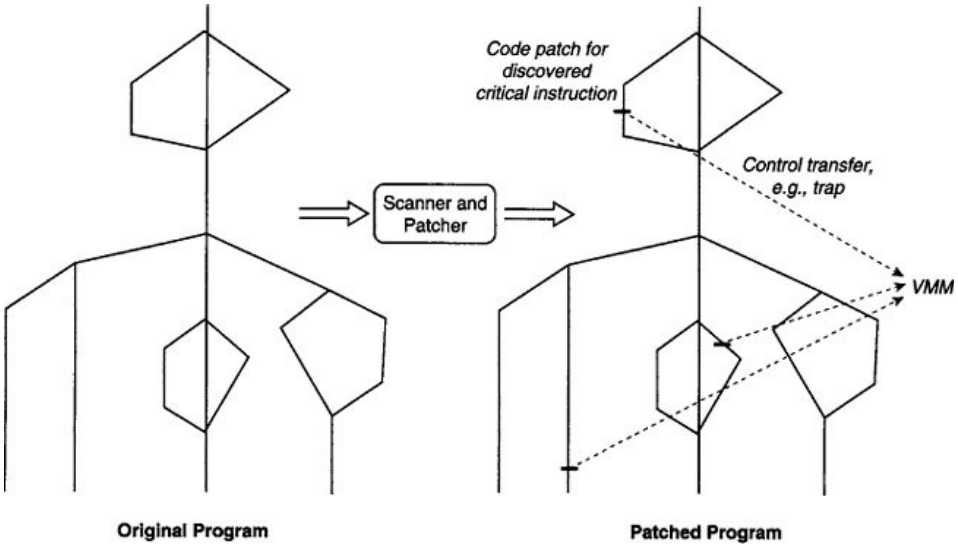



**Figure 8.11** Scanning and Patching Code in a Hybrid Virtual Machine System.

## 8.2.4 Patching of Critical Instructions

As noted earlier, the problem of discovering critical instructions has a lot in common with the general problem of dynamic binary translation discussed in **Chapter 2**. The basic idea in both cases is to locate sections of code about to be executed and perform transformations on them. A major difference is that in binary translation, both the target code and the runtime execute in user mode, and hence a switch between them can be done via a jump. In the VM context, there is an actual change of privilege mode, which means that the jumps to the runtime are replaced with system calls (traps) to the VMM.

One way to discover critical instructions is for the VMM to take control at the head of each guest basic block and scan instructions in sequence until the end of the basic block is reached. If a critical in-

quence until the end of the basic block is reached. If a critical instruction is found during this scan, it is replaced with a trap (system call) to the VMM, and the critical instruction at the location is saved in a VMM side table, along with its original address. Another trap back to the VMM is placed at the end of the basic block in order to allow the VMM to regain control when the block completes execution so that it can scan and patch the next basic block to be executed. If the next block has already been patched, the VMM simply jumps to the next block. This method ensures that every basic block to be executed is patched first.

To reduce overhead, the trap at the end of a scanned basic block can be replaced by the original branch or jump instruction after all possible successor basic blocks have been patched; this is analogous to chaining basic blocks in a code cache (**Section 2.7.1**). For example, if the basic block ends with a conditional branch that has been replaced with a trap, the basic blocks located at both the taken and the fall-through addresses can be scanned the first time the trap is encountered. After patching any critical instructions contained in the successor blocks, the trap is replaced with the original conditional branch. This technique is relatively simple with direct branches, i.e., branches that have a target specified as an offset relative to the address of the branch instruction itself. For indirect jumps and branches where the destination address is located in a register, the trap typically cannot be replaced, because of the difficulty in predicting all possible target addresses.

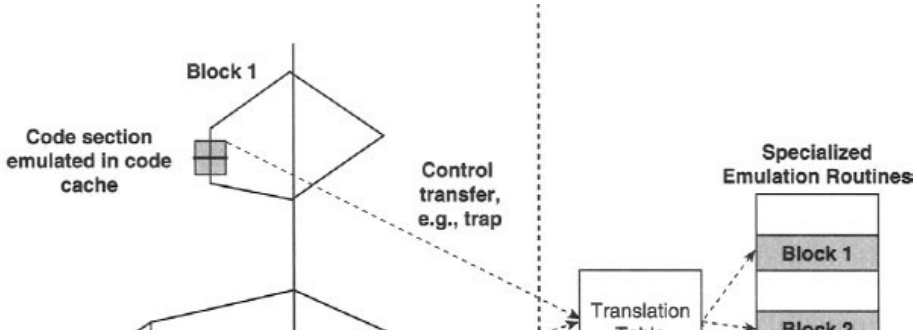
In order to further reduce overhead, instead of scanning single basic blocks at a time, the VMM can scan through all branches that allow target computation. Thus for a conditional branch, both the fall-through as well as the taken branch paths are followed during the initial scan, and this process is repeated until a control-transfer instruction is encountered for which it is not possible to determine all targets. In order to avoid memory paging caused by scanning possibly unused paths, the scanning should be limited to those code

pages already in physical memory.

Observe that a critical instruction appearing in a guest application program (i.e., in the VM's user mode) needs to be patched only if it causes some action to be taken when performed in user mode. For example, if a critical instruction behaves like a no-op in user mode, and if it can be determined that the instruction is to be executed by its VM in user mode, it is not necessary to patch the instruction with a trap. Thus, some of the overhead of trapping can be reduced in the virtual user mode. Unfortunately, as long as the instruction set contains even one critical instruction that, in user mode, performs either a read or a write action on some resource, the instruction must be emulated and the overhead of code scanning and patching must be incurred. Finally, the process of scanning and patching critical instructions is often sensitive to the characteristics of the underlying ISA and is fertile ground for engineering optimizations beyond those described here.

## 8.2.5 Caching Emulation Code

The overhead of VMM interpretation can become a problem when the frequency of sensitive instructions requiring interpretation is relatively high. One way to reduce this overhead is to cache the actions taken during the first interpretation, with a view to avoiding many of the interpretation steps when the same instruction is encountered in the future. This is similar to code caching as described in connection with process virtual machines in **Chapter 3**.



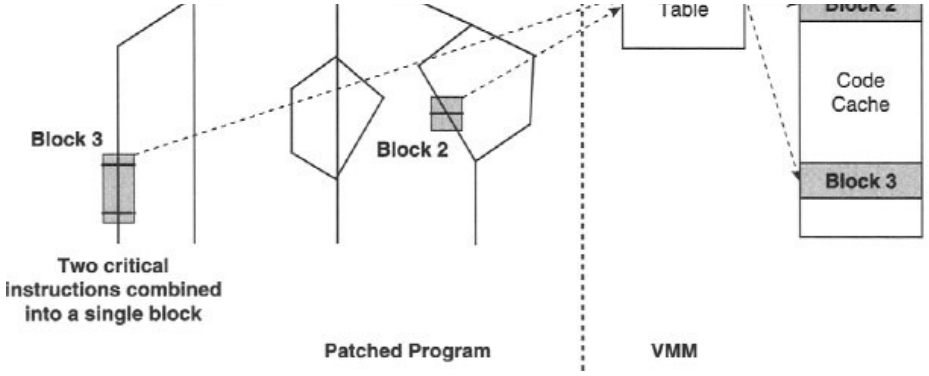



**Figure 8.12** Code Caching to Improve Performance of Emulating Critical Instructions.

Ideally, code caching is done on a block of instructions surrounding the sensitive instruction, since larger blocks lend themselves better to optimization. In contrast to what was shown in **Figure 8.11**, a system call trap may be inserted at an instruction earlier than the critical instruction. As depicted in **Figure 8.12**, the trap causes control to be transferred to the VMM, which then locates the cached emulation code via a table lookup using the address of the trapping instruction. This cached code is an emulation of an entire block of the original code, including the critical instruction, and hence is specific to the location of the trapping instruction. When the VMM returns control to the virtual machine, it restores the program counter to the point immediately after the block that was emulated rather than to the instruction following the trap.

Unlike the simple interpretation case, where there is only one copy of the code that emulates all instances of a specific type of instruction, each instance of the critical instruction in the application will be associated with a distinct piece of cached code. The advantage here is that the cached code can be made specific to the instance and hence can be optimized, often in a way that eliminates a significant fraction of the overhead.

In contrast to process virtual machines, the management of this code cache is somewhat simpler, because code in the cache is executed after the trap, when the machine is already in system mode. The user code can never corrupt or read the contents of the code

The user code can never corrupt or read the contents of the code cache, which is therefore always protected from the user application. On the other hand, self-modifying code remains a problem, and mechanisms are needed to intercept modifications of the original code—such modifications may necessitate invalidations of parts of the code cache.

## 8.2.6 Efficient Virtualizability of Common Instruction Sets

The System/370 architecture provides two modes of operation, a *user* mode and a *privileged* mode (what we have been referring to generally as *system mode*). The operating system is expected to run in the privileged mode. Some of the instructions in the instruction set are privileged; they perform the specified action only when the processor is in privileged mode. These privileged instructions typically manage protected resources in the processors. In order to prevent either malicious or accidental modification of these resources, the privileged instructions trap when executed in the user state.

The VMM for System/370 (called the control program, or CP) has to run in the privileged mode in order to be able to allocate resources to different virtual machines at different times. The virtual machine is created in the privileged mode, but it executes in the user mode. This ensures that each virtual machine is isolated from other virtual machines and from the CP itself.

All sensitive instructions on the System/370, in the Popek-Goldberg sense, are also privileged. VM/370 is an example of a virtual machine employing direct native execution.

As noted earlier, an interesting aspect of the System/370 VM project was the development of a single-user operating system that proceeded in parallel with CP development. This operating system, called the Conversational Monitor System (CMS), took advantage of

the CP environment by allowing all applications to run in the "pseudo-supervisor," state with the user being allowed to use any instruction, whether privileged or not. This is safe because CMS and applications execute on a virtual machine protected by the CP from wantonly affecting the real resources of the computer. The ability to use privileged instructions in their code allows users to experiment in many ways, e.g., to add input/output devices not allowed by the standard system. **Figure 8.13** shows a comparison of the handling of privileged instructions at various levels in various schemes. This feature of the CP/CMS system is highlighted in **Figure 8.13**c.

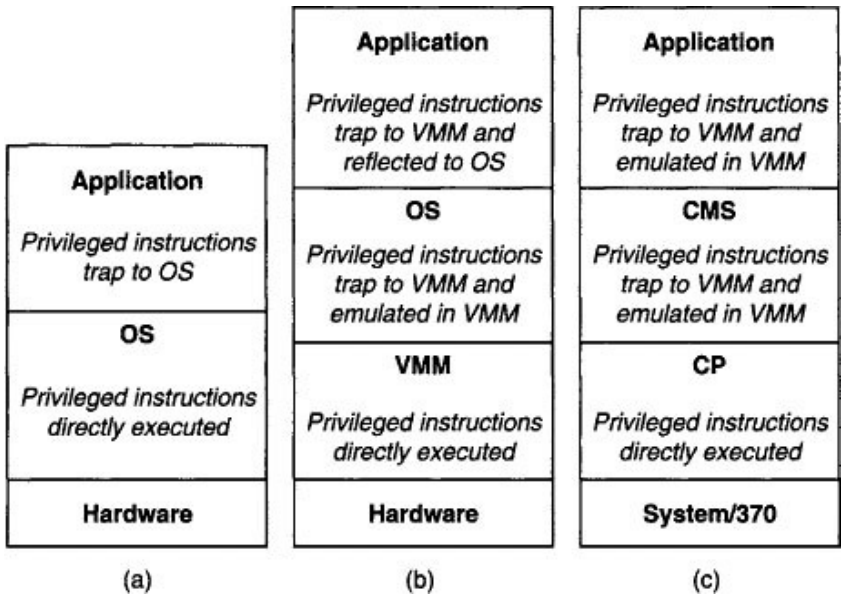



**Figure 8.13** Comparison of the Execution of Privileged Mode Instructions on a VM/370 with that on a Native System and Other Virtual Machine Systems. *(a) A standard system; (b) a virtual machine system; (c) the CP/CMS system.*

Despite the fact that most of the concepts in this chapter have been known for at least 30 years, many ISAs developed during those 30 years do not satisfy the sufficient conditions of Theorem 1. For example, the most commonly used ISA today is the Intel IA-32, and the IA-32 contains 17 critical instructions (**Robin and Irvine 2000**). The likely reason for this is that system VMs have not been considered important because of the tremendous decrease in the cost of computer systems, which makes it easier for each user to have his/her own machine. However, in recent years there has been resurgence

of interest in VMs because of the importance of security in a networked environment and the desire to provide cross-platform compatibility of major software packages. Furthermore, large, expensive server systems are shared among many groups of users, and so several of the original VM motivations remain.

# 8.3 Resource Virtualization—Memory

In some respects, a virtual machine can be considered a generalization of the concept of virtual memory that preceded it. Virtual memory makes a distinction between the logical view of memory as seen by an application programmer and the actual hardware memory resource as managed by the operating system. Appendix A presents a discussion of the basic concepts underlying virtual memory. Also included is a discussion of the structures, e.g., page tables, needed to support virtual memory. It turns out that this hardware support is also sufficient for providing each guest virtual machine in a virtual machine system, with the view of having (and managing) its own "real" memory, which, in fact, is an illusion created by the underlying VMM.

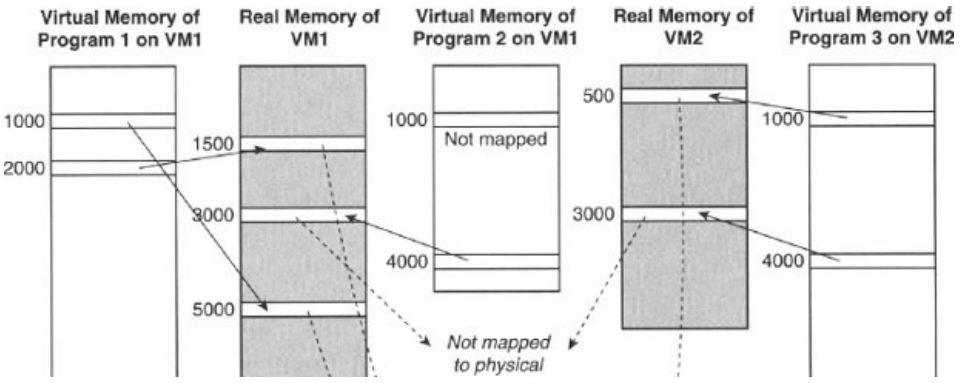
## 8.3.1 Virtual Memory Support in a System Virtual Machine Environment

In a system VM environment, each of the guest VMs has its own set of virtual memory tables. Address translation in each of the guest VMs transforms addresses in its *virtual* address space to locations in *real* memory—this real memory would correspond to the *physical* memory on a native platform; in a VM platform, however, this is not the case. Rather, in a system VM environment, a guest's real memory address must undergo a further mapping to determine the address in physical memory of the host hardware. Note that here we

are drawing a clear distinction between *real* memory and *physical* memory—terms often used interchangeably in other contexts. Physical memory is the hardware memory. Real memory is a guest VM's illusion of physical memory; an illusion supported by the VMM when it maps a guest's real memory to physical memory. It is this real-to-physical mapping that implements the *virtualization* of memory in a VM system.

The addition of another level in the memory hierarchy necessitates additional mechanisms for memory management. Note that the combined total size of the real memory of all the guests could be (and often is) bigger than the actual physical memory on the system. Hence the VMM maintains a swap space, distinct from the swap spaces of each of the guests. The VMM manages physical memory by swapping guest real pages into and out of its own swap space.

**Figure 8.14** shows a set of page table maps belonging to two different guest VMs in a virtual machine system. Each entry in a *page table* maps a location in the virtual memory of an application to a location in the guest VM's real memory. In order to convert a real page address to a physical address, the VMM maintains a *real map table* mapping the real pages to physical pages. In the figure, the physical page frames numbered 500, 1000, and 3000 are assigned to two real pages of VM1 and one real page of VM2. The remaining physical pages may be allocated either to other virtual machines or to the VMM itself.



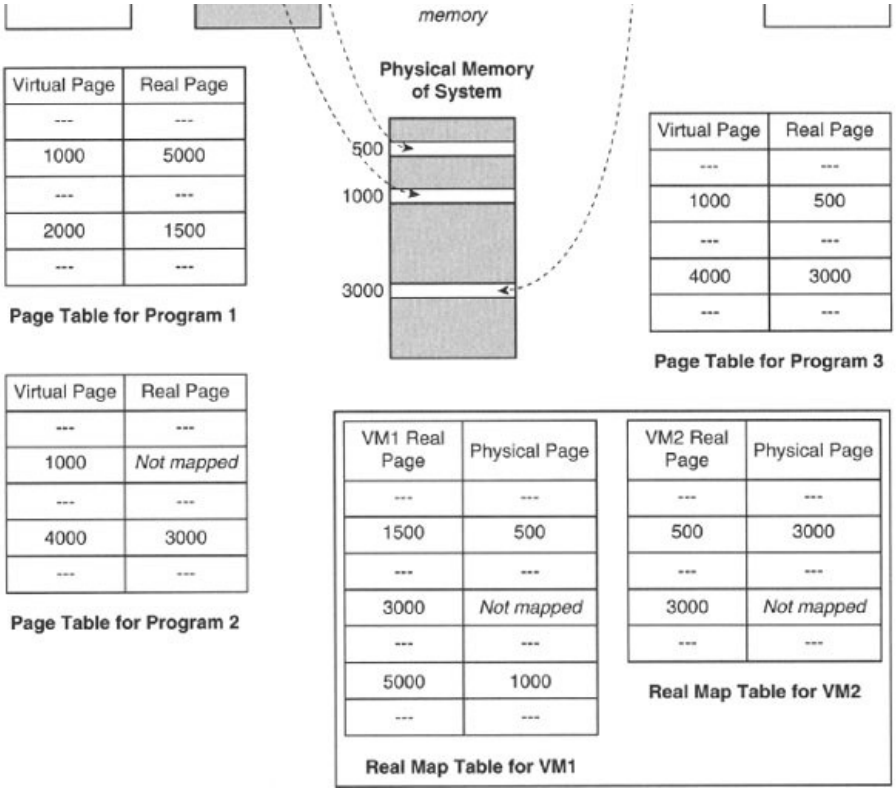



**Figure 8.14** Memory Mapping of Guest VMs. *Virtual pages are mapped to real pages by page tables maintained by each virtual machine. Real pages are mapped to physical pages, using tables maintained by the VMM (at the bottom of the figure). Observe that the real memory of each VM can be larger than the physical memory—the real memory of VM1, for example, is as large as its virtual memory.*

On contemporary platforms, page translation is supported by a combination of a page table and a translation lookaside buffer (TLB); see Appendix Section A3.4. Depending on the ISA, either the page table or the TLB is architected. If the page table is architected, then its structure is defined by the ISA, and the operating system and hardware cooperate in maintaining and using it. In this case, the TLB is maintained and used only by the hardware; it is not visible to the operating system. When there is a TLB miss, the hardware "walks" the architected page table to find the appropriate entry to place in the TLB; if the entry is not mapped to physical memory, there is a page fault and the operating system takes over.

On the other hand, if the TLB is architected, then its structure and special instructions to manipulate it are part of the ISA; the page tables are part of the operating system implementation. Hardware is