In the IEF mode, most privileged instructions are executed directly in hardware. The notable exceptions are I/O instructions and some of the more complex, though infrequent, instructions. Once the IEF gets control, it starts executing the instruction stream from the virtual machine, directly executing many of the instructions and interpreting others in hardware. As shown in **Figure 8.20** there are two ways in which the hardware interpretation of instructions may be suspended and control returned to the software part.

1. An interrupt may be generated that is determined by the IEF hardware to be meant for either the VMM or another virtual machine. A return from such an interrupt points to the SIE instruction that was executed by the VMM.

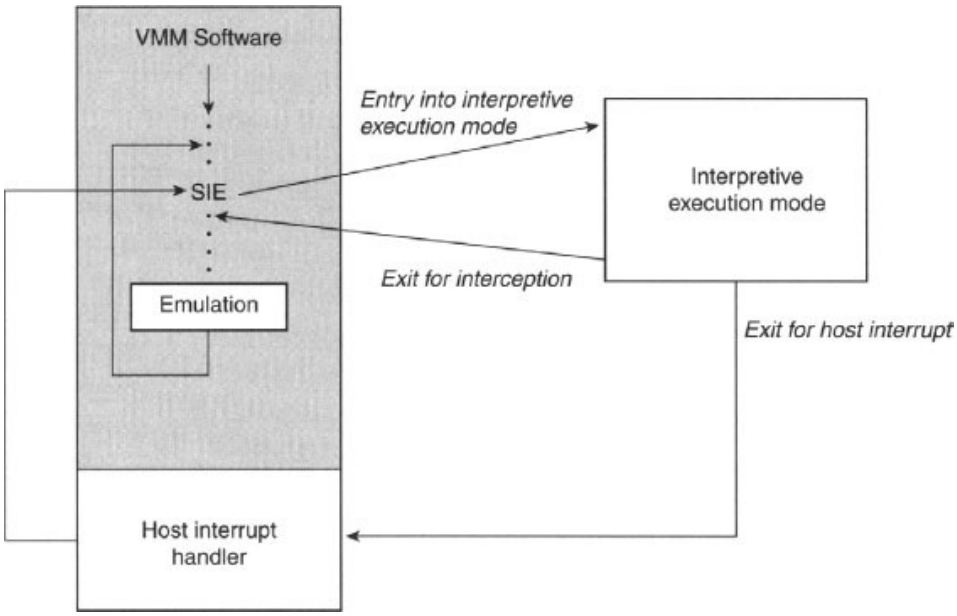


Figure 8.20 Interpretive Execution Entry and Exit. *Execution of an SIE instruction causes the processor to enter the interpretive execution mode. Exit from this mode can occur either because an interrupt needs to be handled by the host or due to one of several conditions that require servicing by the VMM. In the former case, the machine returns back to the interpretive execution mode. In the latter it proceeds to the next instruction after the SIE.*

2. Control can pass to the instruction following the SIE instruction. This is referred to as *interception.* Interception occurs because of the following reasons.

a. An instruction is encountered whose interpretation is not supported in hardware.
b. An exception occurs during the execution of an interpreted instruction.
c. An instruction is encountered in a mode that forces it to be handled in software, as in a traditional virtual machine system.

d. An externally set interception condition is detected.
e. Some special case is detected; e.g., a guest operating system has entered the wait state.

Before control returns to the software part of the VMM, the IEF hardware must update the table containing the state description of the virtual machine so that it can resume operation at a later time. It must also store some information about the reason for giving control back to software, including parameters that would be needed for software emulation.

TABLE 8.1 Improvement in Performance of Virtual Machines Resulting from the Introduction of Hardware VM Assists on the IBM System/370 (MacKinnon 1979)

		Model 135		Model 145		Model 158	
		DOS/VS	VS1	DOS/VS	VS1	VS1	VS2
Elapsed time (seconds)	Native	2788	3035	2150	1418	1386	572
	Virtual Machine	8172	11598	4520	4089	3769	2696
	Virtual Machine with VM Assist	4226	4063	2723	2024	2004	1149
Relative Batch Throughput	without VM Assist	0.34	0.26	0.48	0.35	0.37	0.21
	with VM Assist	0.66	0.75	0.79	0.7	0.69	0.5
Reduction in supervisor state time through VM Assist		74%	89%	73%	86%	82%	69%
Reduction in elapsed time through VM Assist		48%	65%	40%	51%	47%	57%
Reduction in total number of instructions simulated by VM/370 through VM Assist		87%	95%	86%	94%	91%	74%

For good performance, the IEF relies on many of the hardware assist features described earlier in this section. By having special hardware assists for the timer and for the time-of-day clock and by using bypass techniques to reduce the overhead of memory address translation, the IEF comes closer to its goal of making the behavior of a program on a virtual machine very similar, only slightly slower, compared to its performance on a native platform. This is evident from **Table 8.1** (MacKinnon 1979), which shows the improvement in performance through the use of assists for three different implementations of the System/370 architecture and for three different guest operating systems: DOS/VS, OS/VS1, and OS/VS2 SVS. It can be observed that the difference in performance of a program running natively versus in a virtual machine decreases significantly through the addition of special hardware features that support virtualization.

The interpretive execution facility has evolved considerably since its original introduction. In essence, more and more function was added in hardware to eliminate most of the overhead involved in supporting virtual machines. A comprehensive discussion of the various new features can be found in **Osisek, Jackson, and Gum (1991)**.

8.6 Case Study: VMware Virtual Platform

A popular virtual machine infrastructure for IA-32-based PCs and servers is the VMware Virtual Platform (**VMware 2000**). As mentioned earlier, the VMware system is an example of a hosted virtual machine system. More recently, VMware has included a native virtualization architecture embodied in a product called the VMware ESX Server, which is claimed to provide better resource control, scalability, and performance, but at the expense of full support for all types of hardware. Our discussion here is limited to the hosted system, which has been renamed the VMware GSX Server (**VMware 2001**). We outline next the various reasons that motivated this development of a hosted system rather than a native system.

In comparison to a native VM infrastructure for a large mainframe, such as the VM/370, the development of a VM infrastructure for a commodity IA-32 environment faces several challenges.

- The IA-32 grew up as an inexpensive microprocessor, never intended to be used in large systems supporting multiple users. Among other things this led to an ISA that includes a number of features difficult to virtualize efficiently. On the other hand, the IBM VM systems have existed almost from the beginning of the System/370 line. The IBM architecture team had close interactions with the team developing the VMM (because they worked at the same company), and therefore they avoided features that could hamper the development of virtual machines on its platforms.
- The other difference between the current commodity IA-32 environment and the

IBM System/370 environment is the openness of the system architecture, which has led to a proliferation of different configurations of PCs. Thus, while the VM/370 developers had only to understand the system configurations that IBM shipped to its customers, IA-32 VMM developers must anticipate any of the myriad configurations that a multitude of PC system vendors may ship. The situation is made even more daunting by the proliferation of different I/O devices that are available for users to add to their systems.

- VMware, as a separate company from either a hardware developer, such as Intel, or an operating system company, such as Microsoft, faced an additional challenge. It had to ensure that its VMM software can be easily installed and used. In particular it cannot expect its users to wipe out an existing operating system to install VMware software and then reinstall the old operating system over the VMM. This, in fact, directly influenced the architecture of the VMM developed by VMware.

The VMware Virtual Platform presents a view to the user depicted in **Figure 8.21**. At this level it looks similar to the VM view presented to an IBM System/370 user. However, as opposed to the native nature of the virtual machine system of VM/370, the Virtual Platform is a hosted system. It depends on the existence of a host operating system, such as Windows or Linux, running on the bare hardware to perform certain critical functions. As we have seen in **Section 8.4.3**, a principal advantage of a hosted system is that the VMM can avail itself of any I/O device the host operating system supports with relatively little work.

As discussed in **Section 8.4.3**, a dual-mode hosted virtual machine system must have three components for its VMM: (a) the VMM-n, which is the component working in a privileged native mode, (b) the VMM-u, the component that runs as an application on the host operating system, and (c) the VMM-d, the component that enables communication between the VMM-n and VMM-u. On the VMware system, these three components are respectively named VMMonitor, VMApp, and VMDriver. At any given time, the processor executes either in the host operating system environment or in the VMMonitor environment. Transfer of control between the two worlds is facilitated by the VMDriver and involves saving and restoring all user and system visible state information on the processor. As opposed to

and system visible state information on the processor. As opposed to the view presented to the user shown in **Figure 8.21**, the structural view of the VMware system is as shown in **Figure 8.22**.

Even though the Intel IA-32 architecture features four levels of protection, rings 0 through 3, most operating systems on the IA-32 simply lump the kernel and services, including the device drivers, into ring 0. Thus, for all practical purposes, it is adequate to consider the architecture as having a privileged level, largely consisting of code running in ring 0, and a user level, consisting of code running in ring 3.

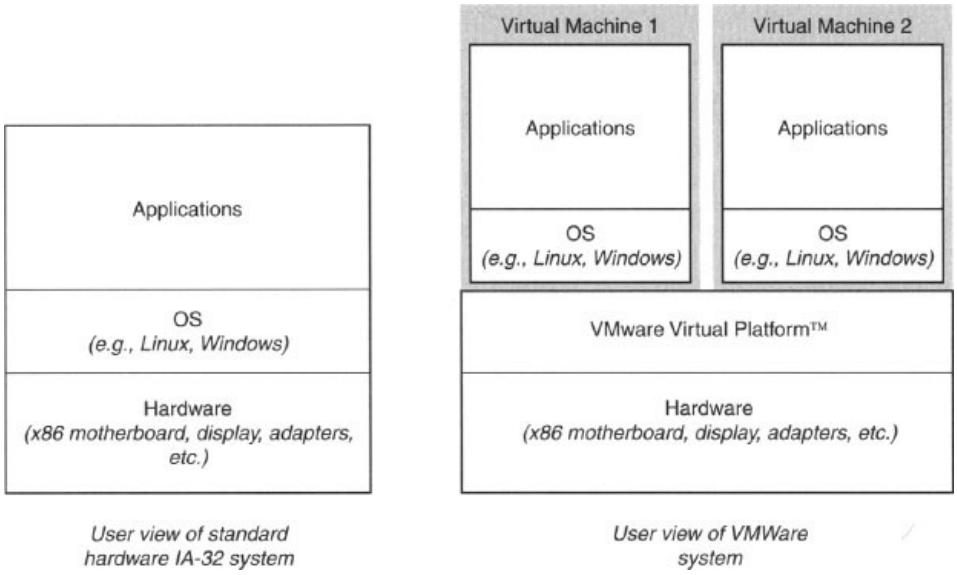


Figure 8.21 User View of the VMware System.

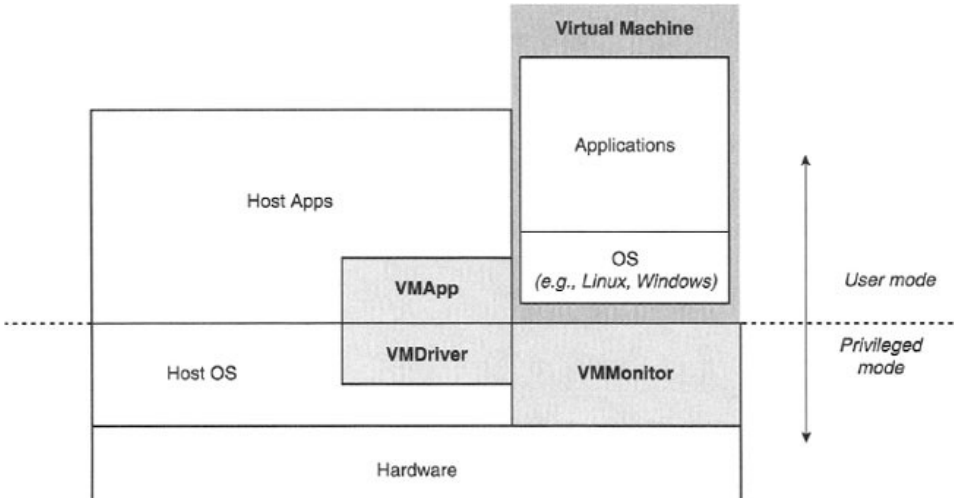




Figure 8.22 Components of the VMware System. *VMMonitor runs in privileged mode. VMDriver is a device driver installed in the host operating system and hence also has system privileges. VMApp runs as a user-mode application on the host operating system.*

VMMonitor operates in privileged mode, but VMApp, which is simply an application that requests services from the host operating system, operates in the user mode. In a sense, VMMonitor can change its personality from being a privileged controller of resources to being just another application on the host operating system. It does this efficiently through VMDriver, a special device driver installed on the host operating system and hence having the same privileges as the host system.

8.6.1 Processor Virtualization

The Intel IA-32 architecture is not efficiently virtualizable, i.e., a VMM cannot be constructed in the strict sense of Theorem 1 in **Section 8.2.1**. Researchers (**Robin and Irvine 2000**) have identified 17 IA-32 instructions that are critical, i.e., sensitive but not privileged. Hence, a virtual machine system for the IA-32 architecture must be a hybrid virtual machine system. VMware presumably includes mechanisms for discovering these critical instructions and for patching (or caching) them, as described in **Sections 8.2.3** and **8.2.4**.

The 17 critical instructions fall into two broad categories.

- **Protection system references:** These instructions reference the storage protection system, memory system, or address relocation system. The hazard here is the possibility of some virtual machine accessing locations not in its virtual memory. An example of this is the MOVE instruction, which moves a value from a general-purpose register to the CS register, the control register that specifies the current privilege ring number in bits 0 and 1. In order to offer some protection, an instruction such as mov ax, cs executed in the user mode disallows the CS register to be loaded but does not generate a trap. Instead of trapping, the instruction generates a no-op and hence is not efficiently virtualizable.
- **Sensitive register instructions:** These instructions attempt to read or change

resource-related registers and memory locations, such as a clock register or interrupt registers. An example is the POPF instruction mentioned in **Section 8.2.1**. This instruction pops a word from the top of a stack in memory, increments the stack pointer by 2, and stores the value in the lower 16 bits of the EFLAGS register. One of the bits in the EFLAGS register is IF, the interrupt-enable flag, which, in the user mode, is not modified when POPF is executed. To understand why this is a sensitive instruction, consider the case where a guest operating system executes a POPF instruction. The operating system may require that the IF bit be changed; but if it is running in the user mode under the virtual machine, the IF bit is not changed. The guest operating system could later take erroneous actions because the flag bit had not been set as expected. In a hybrid virtual machine implementation, such an instruction would be discovered by the VMM and patched to generate a trap. The trap handler in the VMM would then emulate the instruction and achieve the result expected by a guest operating system.

Let us now examine what happens when a guest operating system in a virtual machine attempts to use a critical instruction. The piece of IA-32 code shown in **Figure 8.23** is an example of real code (**Rosenblum 2000**). In this code, the flags are saved on stack using a pushfd instruction before a small piece of code is executed that could potentially change them. When they are restored, the expectation is that the flags will be exactly the same as before being pushed on the stack. In the privileged mode, the popfd would adequately restore the flags to their original state. However, when the code is executed in user mode, as it would be on a virtual machine, not all flags will be properly restored, unless the instruction is emulated.

```
pushfd
cli
mov         cax, (0x824)
cmp         cax, 1
jc          5
mov         (0x900), cdx
popfd
add         cdx, cax
```

Figure 8.23 Sample Intel IA-32 Code Using the popfd Instruction.

In order to perform this emulation, VMMonitor scans the instruc-

tion stream being executed and detects the presence of the popfd instruction in the stream. The monitor then substitutes this instruction with another set of instructions that takes the processor into the privileged state and emulates the action of the original code. The emulation would include a check for the current mode of operation and execute one of two actions for popfd, depending on the mode. Specifically, if the virtual machine itself was supposed to be executing in the system mode, the emulation of the popfd instruction must include the loading of the EFLAGS register or, more accurately, the virtual EFLAGS register belonging to the executing virtual machine. On the other hand, if the virtual machine is not in system mode, the emulation must ensure that certain bits, such as the IF bit, are not modified in the virtual EFLAGS.

The setting of the IF should always be in control of the VMM, so that when a virtual machine is swapped out and another one brought in, the IF is not swapped. For each virtual machine there is a location in memory that corresponds to the EFLAGS register, and the emulation of activity on the EFLAGS register is carried out through instructions operating on the virtual flags in memory, rather than by loading the EFLAGS register and performing the action directly. Emulation of the IF in the EFLAGS register implies that any instruction that reads or modifies this bit must also be emulated. The cli instruction in the example of **Figure 8.23**, which clears the interrupt flag, therefore, is also emulated by VMMonitor.

8.6.2 Input/Output Virtualization

The difficulty of virtualizing I/O in popular IA-32-based platforms was a primary reason that VMware developers chose to implement the monitor in a dual-mode hosted VM style. The PC platform supports many more devices and many more different types of devices than any other platform. This has resulted in a massive number of device drivers written by all segments of the industry, including de-

vice hardware vendors, operating system developers, software developers, and even users. VMware takes a multifaceted approach to solving the I/O virtualization problem (**Sugerman, Venkitachalam, and Lim 2001**).

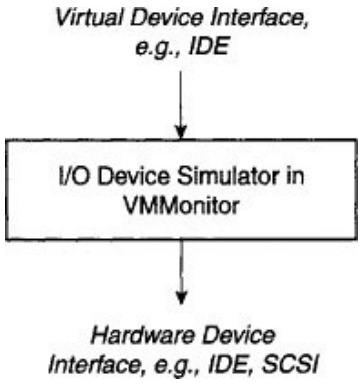


Figure 8.24 Mapping a Virtual Device Interface to a Hardware Device Interface in the VMware Virtual Platform.

Emulation In VMMonitor

The first way to virtualize an I/O device is to emulate the device in the VMMonitor. If the device to be virtualized already has a physical counterpart on the host, the job of emulating it is simply one of converting the parameters in some virtual device interface (VDI) into parameters of the actual hardware device interface (HDI). Thus, the i n and out I/O instructions are intercepted by the VMMonitor and converted to appropriate i n and out instructions for some actual physical device (**Figure 8.24**). A situation where such a scheme would be used is that of converting a command to read a block from an IDE drive to one that reads a block from a SCSI drive. Unfortunately, such conversion requires some knowledge of the ports and memory map locations associated not only with the SCSI drive, but also with the IDE drive supported in the guest operating system.

Using the Services of the Host Operating System

The VMware developers discovered that while some interfaces, such

as the IDE interface, appear to be well standardized across the industry, there is a noticeable variation in devices following the SCSI, LAN, or graphics standards. This called for a different approach to emulating these interfaces. Instead of converting the in and out I/O instructions to new in and out instructions, they are converted into a set of library service requests for the host operating system, which invariably already supports the needed device drivers.

This technique is illustrated in **Figure 8.25**. The method relies on the dual personalities of the Virtual Platform. The VMMonitor contains device models for each of the virtual devices supported. When a guest I/O instruction is encountered, it traps to the VMMonitor code that alters the state of the device model corresponding to the requested device. If the requested device is not natively supported by VMMonitor but is supported by the host OS, the request is converted into a host OS call. For example, if the host operating system is Windows NT, it is converted into a *win32* system call. The VMMonitor, through VMApp, now acts as a well-behaved application under Windows NT. When the application returns from this system call, the control gets back to the VMMonitor and then into the application running on the virtual machine. VMMonitor uses this approach for virtualizing the floppy disk drive, the CD-ROM drive, the sound card, the serial port, and the parallel port. In measurements taken on a real system, the overhead of switching from one mode to the other was determined to be only tens of microseconds.

The dual personality assumed by the VMMonitor confers advantages beyond reducing the complexity of supporting a large variety of I/O devices.

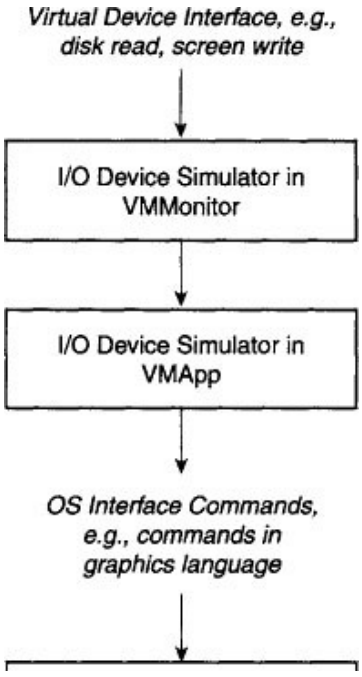
- As long as the operating system provides a reasonable suite of services to perform I/O, any operating system can be used as the host. All devices supported by that operating system would automatically be available to VMMonitor.
- The VMMonitor need not limit its access of the host operating system to its I/O features. All services available on the host operating system can be taken

features. All services available on the host operating system can be taken advantage of because VMMonitor installs on the host operating system just like any other application. In particular, it can use the file system services of the host, say, to emulate disks.

- When making a transition from an old operating system to a newer one, it may be beneficial to use the old operating system as the host, with the new one working in a virtual machine. Performance-critical applications can be run on the host directly during the transition period.

New Capability for Devices Through Abstraction Layer

VMApp has the ability to insert a layer of abstraction above the physical device. This allows the incorporation of new functionality on the device that may not have been possible or convenient with the original device. For example, the disk on a virtual machine in VMware can be treated either as a raw disk or as a file on the host operating system. In the latter case, it then becomes possible to have "undoable" disks, where recent actions taken on the disk can be undone. This also makes it possible to save or discard entire sessions by providing a commit feature that performs explicit commits of disk writes at the end of the session. Such a capability is particularly useful, for example, for a test session or if one needs to discard a session because of an insecure access.



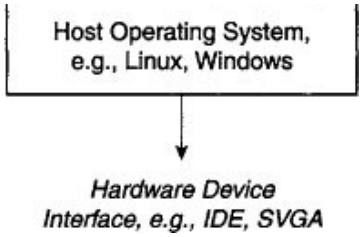


Figure 8.25 Use of the Operating System Library Function Interface to Map a Virtual Device Interface to a Hardware Device Interface.

This abstraction layer can also provide a way to reduce performance losses due to virtualization. For example, a virtual Ethernet switch is provided in VMware between a virtual NIC (network interface controller) and a physical NIC to allow multiple virtual machines to share a physical NIC, similar to what was described in connection with **Figure 8.19**. Different IP addresses can be given to each of the virtual machines. The virtual machines sharing the physical NIC can communicate with each other through this virtual LAN, exactly in the manner of a more traditional LAN interconnection between physical machines. The latency of communication between these virtual machines can actually be made lower than that encountered in a physical system because the LAN is emulated using special communication paths that are local and more efficient than the standard Ethernet path.

A different type of abstraction is exemplified by the provision in VMware for an alternative user interface. In general, user interface controls such as the video card, the keyboard, and the mouse are physical resources that are virtualized in a manner similar to other physical resources, such as general-purpose registers. A user on a virtual machine takes direct control of these devices just as he/she would on a real machine. Thus the entire screen of the monitor can be under the control of the virtual machine. In addition, VMware also provides the capability to make the user-interface display appear as a window on the user interface of the host operating system, e.g., as one of the windows of a Windows desktop. This capability allows the user to access applications on both platforms in a convenient way. Using special communication paths between VMMonitor

client way. Using special communication paths between VMMonitor and the host operating system, it is even possible to cut and paste between a window on the host and an application on a virtual machine.

8.6.3 Memory Virtualization

VMMonitor virtualizes the physical memory of a virtual machine by using the host operating system to allocate or release the real machine's physical memory. The guest operating system does standard demand paging, though physical memory pages for the virtual machine are assigned by the host operating system. Paging requests are not directly intercepted by the VMM. They are converted into disk read/writes by the guest OS exactly as they would be on a real machine. These disk read/writes are translated by VMMonitor to requests on the host operating system through VMApp. The requests are actually made to appear by VMDriver as large DMA requests on the host operating system. The activity of applications on the host determines whether or not such a DMA request is satisfied from the memory of the host or results in paging on the host. The standard replacement policies of the host could end up replacing critical pages, thereby degrading performance of the virtual machine system in the presence of other host applications. In order to alleviate this problem, VMDriver pins some of the critical pages of the virtual memory system in physical memory, especially those pages belonging to the VMMonitor's active working set.

8.7 Case Study: The Intel VT-x (Vanderpool) Technology

In describing the VMware Virtual Platform in **Section 8.6**, we indicated several of the problems in virtualizing the Intel IA-32 architecture. We also saw how the VMware GSX server tackled the virtualizability problem to provide virtual machine capability on this popu-

lar platform using a hosted VM system. However, the various techniques used to finally achieve virtualization come at a cost—complexity of code and performance overhead. Some of the overhead, particularly that of scanning and patching the code to get around the presence of critical instructions in the instruction set, will also be present in native virtual machine implementations like the VMware ESX server. This was the motivation behind the current trend towards paravirtualization, where the interface presented by the virtual machine is not identical to that of the architecture of the underlying processor, but rather simplified to eliminate the effect of critical instructions as much as possible.

The performance overhead of handling the sensitive instructions in the VMM still remains, even with paravirtualization. In **Section 8.5** we described various ways to speed up virtualization through hardware enhancements of the processor. VM Assists and the Interpretive Execution (IE) capability went a long way in improving the performance of the z/VM system on the IBM zSeries mainframes. Intel recently developed a capability, the VT-x, or Vanderpool, technology for IA-32 processors, that is conceptually similar to the IBM's IE capability, and promises to enhance the performance of IA-32 virtual machine implementations.

The main feature of VT-x is the inclusion of the new VMX mode of operation. When in VMX mode, the processor can be in either VMX root operation or VMX non-root operation. In both cases, all four IA-32 privilege levels (rings) are available for use by software. VT-x, in effect, provides four new less privileged rings of protection for execution of guest software, in addition to the usual four rings for use by the VMM. The behavior of the processor in VMX root operation is largely similar to its function in a normal processor not incorporating the VT-x technology, the main difference being the inclusion of a set of new instructions called the VMX instructions. The behavior of the processor in non-root operation is limited in some re-

spects from its behavior on a normal processor in order to support virtualization. The limitations are such that critical shared resources are kept under the control of a monitor running in VMX root operation. This limitation of control of resources extends also to non-root operation in ring 0, which, in normal processors, is the most privileged level. Thus the intention is for the VMM of a virtual machine system to work in VMX root operation, while the virtual machine itself, including the guest operating system and applications, works in VMX non-root operation. Because VMX non-root operation includes all four IA-32 privilege levels (rings), guest software can run in the rings in which it was originally intended to be run (i.e., the guest operating system kernel can run in ring 0 and guest applications can run in ring 3).

8.7.1 Technology Overview

The typical sequence of operations in a virtual machine system is shown in **Figure 8.26**. A processor which has been turned on in a normal mode can be made to enter VMX root operation by issuing a vmxon instruction. The virtual machine monitor running in root operation sets up the environment for each virtual machine and initiates the virtual machine by issuing a special instruction, vmlaunch. Various situations, especially attempts to reconfigure critical resources shared with other virtual machines or with the VMM, cause the processor to relinquish control to the VMM in root operation. The processor may also exit from the virtual machine explicitly through a vmcall instruction. Re-entering a previously launched virtual machine is accomplished through the lower-cost vmresume instruction, which puts the processor back in non-root operation. Thus, in the steady state, the processor should be working largely in VMX non-root operation with occasional short excursions to the VMM. Finally, when the processor wishes to leave VMX operation, it will bring down each of the virtual machines in turn and return to the normal mode by issuing a vmxoff instruction.

This is a high-level view of how a virtual machine system can be created on a VT-x enabled IA-32 processor. There are many aspects which need to be taken care of in launching and exiting from a virtual machine; the interested reader is referred to Intel's Preliminary Specification document (**Intel 2005**) for more details on this topic. We limit our discussion below to the concepts underlying the technology.

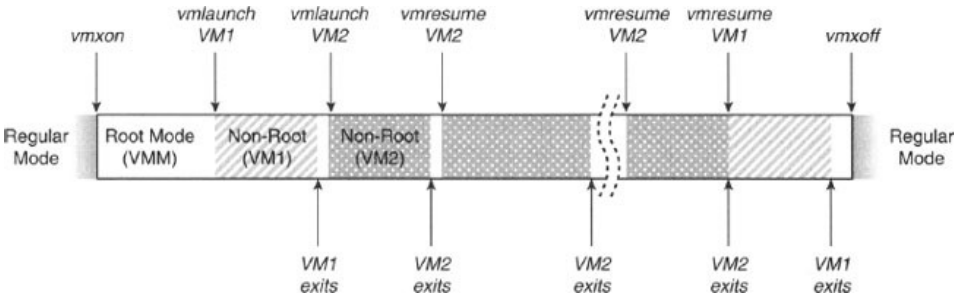


Figure 8.26 Transition Between Operation Phases with the Intel VT-x Technology. *White areas represent the operation in root mode of the virtual machine monitor (VMM). The cross-hatched and dotted areas represent the regions in time when the two active virtual machines are executing.*

8.7.2 Capabilities of the Technology

A key aspect of the VT-x technology that allows faster virtual machine systems to be built is the elimination of the need to run all guest code in the user mode, essentially by providing a new mode of operation specifically for hypervisors and VMMs. For code regions that do not contain instructions that affect any critical shared resources, the hardware executes as efficiently as it would have on a normal machine. Since hardware keeps track of the virtual machines that are active, even larger regions can be executed at full speed by temporarily mapping the shared resources to the corresponding resources in the virtual machine. It is only in the few cases where this is not possible that a certain degree of emulation must be performed by the VMM. Thus, once in the virtual machine, the exits back to the monitor are far less frequent in the hardware case than in software virtualization.

The other major source of overhead in a software-based solution is the maintenance of state information. Various tricks in the software, especially mapping the frequently used registers to hardware register elements, reduce the overhead of maintaining and changing state. VT-x technology provides hardware that allows almost all of the state-holding data elements to be mapped directly to their native structures during virtual machine execution. Moreover, the technology provides a formal data structure called the VMCS (virtual machine control structure) that encapsulates all the information needed to capture the state of a virtual machine or to resume a virtual machine. The hardware implementation can take over the tasks of loading and unloading the state from their physical locations, without requiring the VMM to perform expensive load and store operations on large quantities of data.

The VT-x hardware is thus able to reduce the overhead in a classic native VM system considerably. It also eliminates the need for paravirtualization, thereby allowing more standard versions of operating systems to be used in virtual machine systems. The vmcall instruction, which can be used by a VMX non-root guest to communicate with the root mode VMM, can be used as a mechanism to pass hints and data that can help the VMM to perform more intelligent allocation of resources.

VT-x can be adapted also to construct a hosted VM system. Since VMX root operation has been designed to be as close as possible to the normal operation of a processor, it is possible to take a standard implementation of a host OS and make it run in VMX root operation. The VMM, also running in VMX root operation, is then in the same environment as the host OS, and has the same privileges as the host OS, just as in a typical hosted-VM implementation like VMware's.

An important reason for the choice of a hosted system for the VMware GSX server was the need to virtualize the large number of existing devices and device drivers that exist in the IA-32 PC world.

This need still remains; the VT-x technology does not address I/O virtualization. It must be noted also that the VMX root operation is not exactly identical to normal operation of the processor. There are restrictions imposed in VMX operation, such as the inability to reset the CRO.PE and CRO.PG bits to 0, which prevent a host OS in VMX operation from running in unpaged protected mode or in real-address mode. Modern applications will not suffer because of this limitation; however, on legacy systems these modes may have to be supported either through emulation in the VMM, or by executing in the normal mode after leaving VMX operation through a vmxoff instruction.

8.7.3 Maintenance of State information

We now turn our attention to the mechanics of state maintenance in the VT-x technology. As mentioned before, the state of a virtual machine is maintained in the VMCS data structure. This structure is fully specified, with the various fields precisely defined. The VMCS can be manipulated only by hardware, or by software running in root operation. The VMCS corresponding to the currently executing VM is identified by a pointer, the VMPTR, which contains the 4KB-aligned physical address of the VMCS. Multiple VMCS structures corresponding to different virtual machines can be active, though only one can be executing at a time on a given logical processor. (There may be more than one “logical” processor in a multithreaded system.) The VMPTR can be modified using the vmptrld instruction in root operation. Similarly, the contents of the VMCS can be accessed only in root operation, and only through vmread and vmwrite instructions. Regular memory load and store operations may not be used to access data in the VMCS because the format used to store the VMCS is not architected, and may change from one implementation to the next.

The different types of information stored in the VMCS are shown in

Table 8.2. The state information that is stored for the guest and the host goes beyond the conventional architected state and includes items like the architecturally hidden part of each segment register. This is needed to maintain a behavior of a virtual machine similar to its behavior on a normal machine. The various control fields determine the conditions under which control leaves the virtual machine (VM exit) and returns to the VMM, and define the actions that need to be performed during VM entry and VM exit. For example, since the MSRs on a processor are implementation-dependent, one of the control fields determines how many model-specific registers (MSRs) must be saved on VM exit, and another field determines the physical address where these registers must be saved.

TABLE 8.2 Information Stored in the Virtual Machine Control Structure (VMCS)

State Area	Guest State	Register State
		Interruptibility State
	Host State	Register State
Control Area	VM Execution Controls	Pin-based Execution Controls
		Processor-based Execution Controls
		Bitmap Fields
		etc.
	VM Exit Controls	Control Bitmap
		MSR Controls
	VM Entry Controls	Control Bitmap
		MSR Controls
		Controls for Event Injection
VM Exit Information	Basic Information	VM-Exit Information
		Vectoring Event Information
	Other Exit Information	Due to Event Delivery
		Due to Instruction Execution

An important part of the VMCS area is the VM-exit information area. This area contains fields that inform the VMM the reason for an exit, as well as other information needed to service the event that caused the exit.

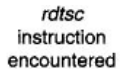
Instructions may cause exits, either unconditionally, e.g., instructions disallowed in non-root mode, or conditionally, e.g., a rdtsc instruction, which causes an exit only if a special control bit “RDTSC Exiting” has been set. There are several such control bits described

in the specification document (**Intel 2005**), that essentially allow the VMM to set up virtual machines with different kinds of behaviors.

8.7.4 An Example: The rdtsc Instruction

We conclude this section by demonstrating how the hardware supports a virtual machine executing the "Read Time-Stamp Counter" instruction, rdtsc. According to the IA-32 architecture, the time-stamp counter is maintained in an MSR called the IA32_TIME_STAMP_COUNTER. Execution of the rdtsc instruction causes the 64-bit value in this counter to be transferred to a specified pair of general-purpose registers. However, if the TSD (Time-Stamp Disable) bit of control register CR4 is set and the privilege level is not 0, a protection-mode exception will result and the counter will not be read.

A flow chart showing the various conditions that arise is shown in **Figure 8.27**. The first thing the hardware checks is the TSD bit in the physical CR4 register that is mapped to the CR4 of the currently active VM. If the TSD bit in this register is set and the privilege level is not level 0, the operation must cause a protection-mode exception in the guest. The hardware directly examines a bitmap, called the Exception Bitmap, in the Processor-Based VM-Execution Controls area of the VMCS for the currently executing VM. This bitmap is a 32-bit vector that contains one bit for each IA-32 exception. If the bit corresponding to the mentioned protection-mode exception is set, the hardware forces a VM exit to allow the VMM to take appropriate action. Otherwise, the exception is directly delivered to the guest as in a processor operating in the normal mode. Thus the hardware avoids the need for the VMM to intercept the exception, inspect it, and reflect it back to the guest OS, as typically done in software VM implementations.



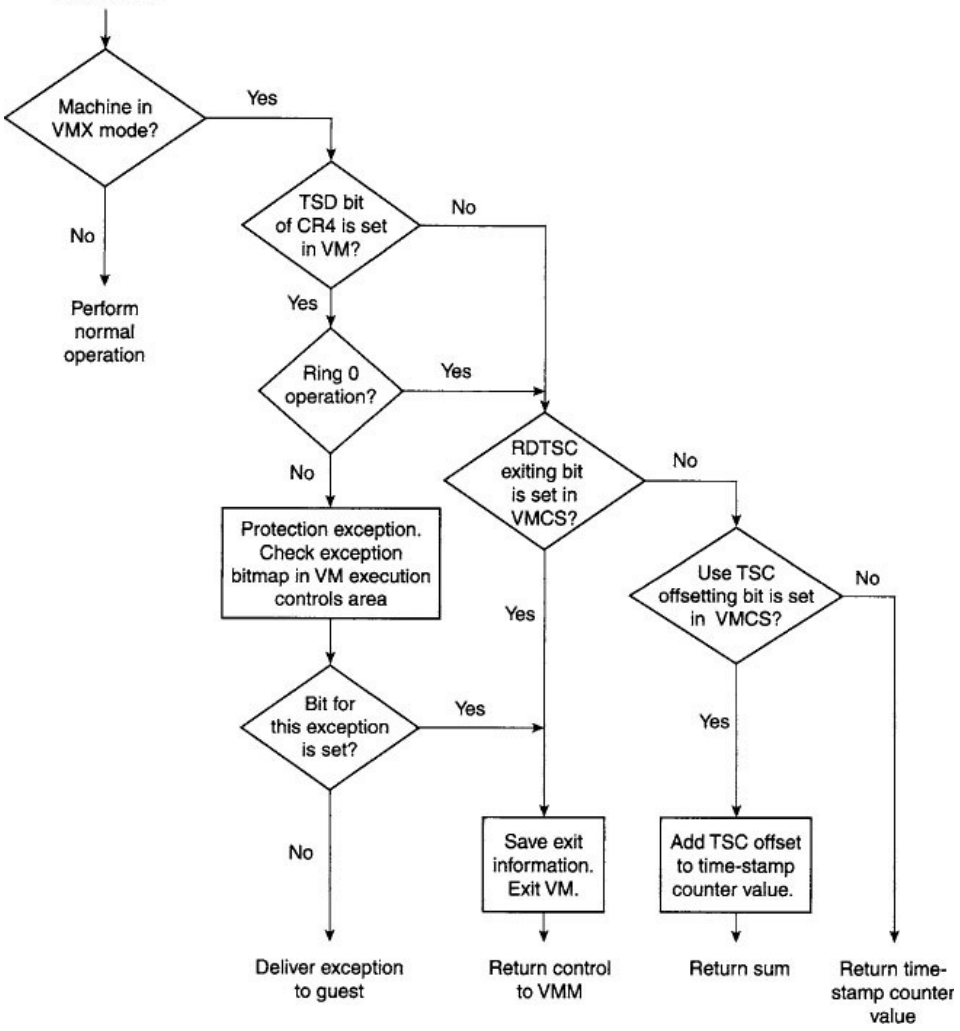


Figure 8.27 Actions Taken by Hardware When a Read Time-Stamp Counter (rdtsc) Instruction Is Encountered.

If the TSD bit is not set or if the privilege level is 0, the hardware checks the "RDTSC Exiting" bit in another field of the Processor-based VM-Execution Control area of the VMCS. If this bit is set, a VM-exit occurs and control returns to the VMM, which can then emulate the desired behavior of the instruction.

If the "RDTSC exiting" bit is 0, the processor checks another bit, the "Use TSC Offsetting" bit, in the same area of the VMCS. If this bit is 0, the value returned is the actual value contained in the physical IA32_TIME_STAMP_COUNTER. When this bit is 1, it is not directly the physical counter value that is returned, but rather the value obtained by performing a signed addition of this value with a value

contained in the “TSC Offset” field of the Processor-based VM-Execution Control area of the VMCS. This mode allows the VMM to approximate different behaviors for a virtual machine. For example, the VMM could use the value in the physical counter at VM-entry and VM-exit to store a TSC offset value such that the processor reports to a virtual machine only the number of cycles elapsed while the virtual machine was active.

This example demonstrates the flexibility provided by the control bits that allows virtual machines with different behavior to be instantiated without incurring a software penalty. And yet, if a more elaborate behavior is needed, a VM-exit can be forced and the desired behavior implemented in the VMM.

8.8 Summary

System virtual machines enjoyed a fair amount of popularity in the 1970s and 1980s, but they appeared to go out of favor in the 1990s as single-user computer systems became affordable. Recently there has been a resurgence of interest in system virtual machines, especially in the Web services area, with the need for massive numbers of simple single-thread systems that seldom have to communicate with each other but that share storage resources. For these applications, virtual machines tend to utilize processor resources better than large clusters of uniprocessors. They also reduce administrative costs by making the process of starting up or shutting down a virtual machine trivial, compared to what would be needed to install or remove a real machine.

While the original popularity of virtual machines was due to the ability for multiple users to use a sophisticated system as though it were their own, the future pervasiveness of virtual machines is likely to arise from other factors. As we will see in **Chapter 10**, there are emerging issues, such as security and system encapsulation, and

emerging models, such as the Grid, that are likely to increase the use of virtual machine technology.

Previous chapter
< Chapter 7: Codesigned Virtual Machines

Next chapter
Chapter 9: Multiprocessor Virtualization >

Chapter Nine

Multiprocessor Virtualization

Many systems today, particularly servers and high-end desktop systems, contain multiple processors. Server systems typically incorporate many processors that share large amounts of memory and I/O devices. Web servers manage huge databases and need to service a multitude of simultaneous requests coming from large numbers of network ports. Computational servers used for large scientific calculations have thousands of processors connected to terabytes of memory and petabytes of disk capacity. Moreover, as levels of integration continue to increase, multiprocessor architectures will soon find their way into laptops and inexpensive desktop systems.

The increasing availability of multiprocessor systems has led to the examination of techniques that can help utilize them more effectively. Often there is a mismatch between the ideal number of processors an application needs and the actual number of physical processors available. With increases in the sizes of multiprocessor systems, it is more often the case that applications cannot exploit more than a fraction of the processors actually available. This may be caused by limitations in the parallelism available in the programs or by limitations in the scalability of applications due to the overhead of communication between processors. This has led to the development of ways in which the multiprocessor system can be *partitioned* so that multiple applications can simultaneously exploit the available resources of the system.

9.1 Partitioning of Multiprocessor Systems

Systems

In this chapter, we discuss general techniques to virtualize multiprocessor systems. As with other virtualization methods, a virtualized multiprocessor gives the appearance of a system that may or may not reflect the exact configuration of the underlying physical system. The term *partitioning* suggests that each virtual multiprocessor system is given a subset of the resources available on the system. The system virtual machine techniques described in **Chapter 8** essentially perform partitioning, in time, of the processor resource. Multiprocessor systems provide a new dimension, that of partitioning of processors in space. Moreover, as we will see, the two can be combined so that a virtual system may have even more processors than the physical number of processors available on the system.

Multiprocessor systems can be configured in a variety of ways, as discussed in Appendix Section A.7. One way is as a cluster system, typically consisting of a number of single-processor (or small multiprocessor) systems communicating with each other through high-speed network interfaces. Another way is as a large shared-memory processing (SMP) platform providing a large number of processors that communicate through shared memory. While there may be reasons to virtualize any of the multiprocessor architectures on any of the platforms, our discussion here will be largely restricted to construction of virtual clustered multiprocessor systems on a host SMP platform, with each node in the cluster being an SMP system. Typically the number of processors needed by a guest SMP system node is small. This is important, because many of the partitioning techniques we discuss assume that the number of processors needed in the guest is not more than the number of available processors on the host. Partitioning therefore provides an illusion of several virtual shared-memory systems operating simultaneously on a single shared-memory host system, as shown in **Figure 9.1**.

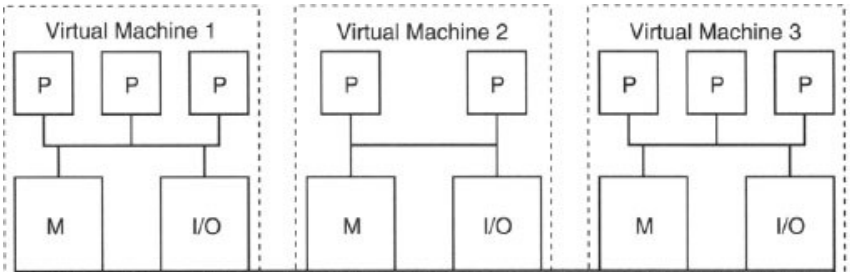
9.1.1 Motivation

Virtualization

Beyond efficiency, there are a number of reasons for virtualizing multiprocessor systems. Some are extensions of the traditional reasons outlined in **Chapter 8**. However, with multiprocessors becoming more common, users are finding additional ways to take advantage of the capabilities offered by virtualization. We outline next the expanded benefits typically provided by virtualizing multiprocessor systems.

Workload Consolidation

The shared-memory multiprocessing paradigm has been used for several years in high-end database servers. This paradigm combines the technological advances made in supporting cache coherence across a large number of processors with the ease of programming afforded by a single namespace. Today, large database servers almost exclusively use shared-memory multiprocessors. The large amounts of memory, the large number of disks, the requirement for high reliability, and the need for special environments driven primarily by cooling and security make these systems very expensive. Yet, for every unit of computation consumed by these large database servers, several more are consumed in other computational layers commonly used in large enterprises. In a three-tier model, for example, there are large numbers of workstations or PCs that feed requests to a first level of servers, commonly referred to as *application servers,* as shown in **Figure 9.2**. The application servers are responsible for running the business process logic but also make accesses to large database servers.



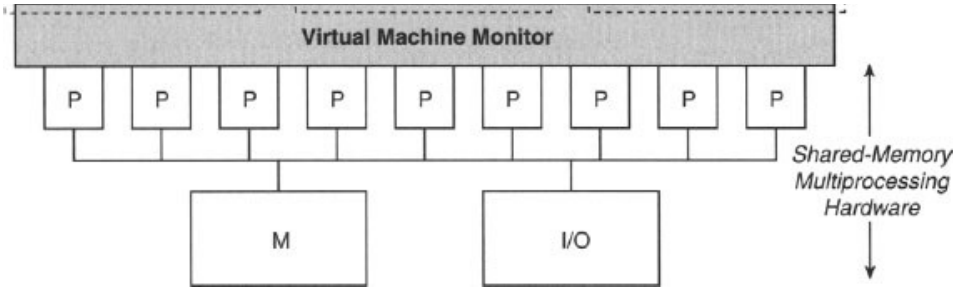
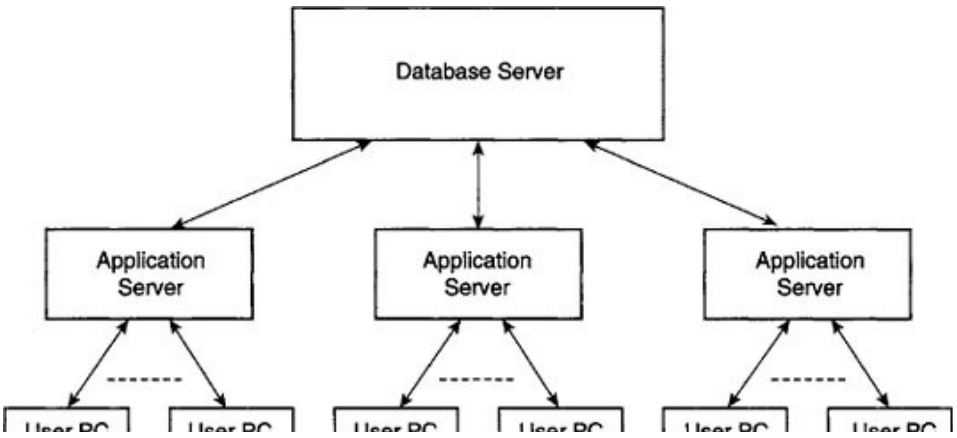


Figure 9.1 Large Shared-Memory System Partitioned into Smaller Shared-Memory Systems.

Administrative costs of computer centers are dependent, to a large extent, on the number of systems that need to be supported. These costs are lowered by reducing the different types of systems in the center. Organizations that already need large servers for their databases find it attractive if there is a way to shift the application server workload to the large servers, as depicted in **Figure 9.2**.

Another important scenario is the consolidation of multiple workstation users on a large remote server. Users of these small systems may be reluctant to migrate to the large system unless they have assurances regarding the privacy of their data and their environment. Moreover, users differ in their preferred operating system and system configuration. Virtualization of large servers through partitioning of physical resources addresses both these concerns—the isolation provided through partitioning satisfies the privacy requirement, while the ability to form variable partitions provides greater freedom to the user compared to a multiprogrammed solution.



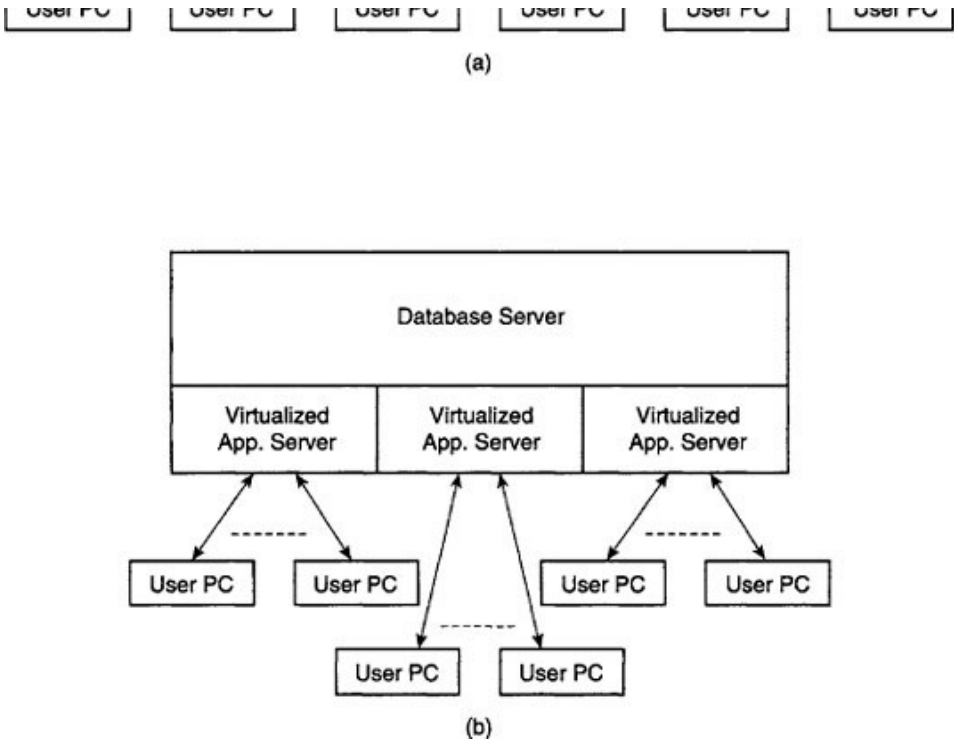


Figure 9.2 Consolidation of Database and Application Servers on a Single Platform. (*a) Typical three-tier server model. (b) Consolidation of the application server with the database server.*

Cluster-Based Programming Model

Large SMP servers provide high performance in many scientific and commercial applications, particularly database applications. On the other hand, there has been a recent proliferation of low-cost cluster platforms, including blade servers. With the rapid increase in programming tools and applications being developed for such platforms, there is increasing pressure to provide the ability to run cluster-based applications on high-end shared-memory systems. Partitioning provides the user of a large shared-memory system with multiple operating system images on multiple smaller virtual multiprocessors that can run such clustered applications. A program written for a cluster using the Message Passing Interface (MPI) (**Pacheco 1996**), for example, can then work unchanged on such a partitioned system. Moreover, the efficiency of execution may be improved by tuning the message-passing libraries to exploit the shared-memory hardware mechanisms available on such a system.

System Migration

Rapid improvements in technology have led to the introduction of new systems with new capabilities at a breathtaking pace. To remain competitive, businesses are often forced to use the latest technology for their computation and server needs. Yet the process of migrating to a new system is arduous. New versions of a given application must be thoroughly tested before they can replace older versions. Similarly, new versions of the operating system or middleware can be used in a production mode only when all applications either have a new version that runs on the new system or when these applications are found to run without problems on the new system. The introduction of a new system is therefore a very disruptive process.

A partitioned system goes a long way toward reducing the pain of moving to a new system, by allowing the testing and verification of the various components to proceed in partitions of the system separated from other partitions that execute production processes. The isolation provided by a partition ensures that any problems discovered during testing do not disrupt production applications by crashing the entire system.

Reduction of System Downtime

Just as with migration to a new system, an upgrade of an existing operating system often involves several installation and configuration steps, which usually can be done only by bringing the system down, i.e., by purging the system of all jobs and users. Having a partitioned system allows system administrators to perform many of the normal downtime activities in a separate partition even when the rest of the system is continuing with production jobs. Thus an upgraded operating system can be completely checked out on a new partition before the rest of the system is brought down and mi-

grated.

Heterogeneous Systems

It is not uncommon for enterprises wishing to change to a new operating system to be forced to continue running some of their older applications on the old operating system. For example, an organization wishing to switch from, say, a proprietary Unix environment to a Linux environment may still need to keep its old environment to run some critical database applications. A platform that supports virtualization would allow both environments to be run on a single server by partitioning the server and running the two operating systems in different partitions.

Improving System Utilization

Most systems are designed to meet peak workloads. Yet the average workload on a typical system is only a fraction of the peak for which it is designed. The ability to partition the system and run a variable number of operating system images allows the system to be configured to changing needs. For peak workloads that need the single system image of a large shared-memory system, for example, the system can be configured to run as a single partition. At other times, the system can be configured for multiple partitions and can be used as a cluster of smaller systems running several applications simultaneously, each on a separate operating system. Such a solution both improves the utilization of the large server and reduces the cost for an enterprise, especially compared to the alternative of buying and operating several servers.

Capacity planning, the process of estimating how much compute power is needed by various workloads in an organization, can help in determining whether partitioning is viable for the organization. Flexibility in scheduling of operations can help in keeping the compute requirements closer to the average workload rather than the

sum of workload peaks.

Resources such as tape drives, optical storage, and high-performance communications adapters are needed by many applications, but typically only for short time intervals. A system might provide as many of these devices as required by the virtual system running in each partition—in fact, this may be a requirement imposed by the operating system running in the partitions, especially when security is a consideration. However, in a large majority of the cases, it is acceptable for the resources to be available only when they are needed by a system—these resources can be "loaned" to other partitions at other times. Common partitioning techniques allow for the allocation of resources in this manner. Dynamic migration of resources from one partition to another goes a long way toward improving the utilization of resources in a system and hence toward reducing the cost of ownership of the system.

Multiple Time-Zone Requirements

It is customary for geographically distributed parts of the same international company to run different operating systems for the different geographical regions, each with its own date-and-time setting. The main reason for this is the need to bring down the system for maintenance or upgrade in each region at a convenient local time, for example, at night, when there are few users loading the system. Server consolidation often finds all parts of the company using the same physical server for their computing needs. By running the workload for each region in a partition isolated from that of the other regions, each region can make decisions independent of the other regions about scheduling its batch workload or its downtime.

Failure Isolation

One of the most important reasons partitioning is becoming popular is its ability to isolate failures. It is not uncommon for today's systems to be vulnerable to either attacks over the network, unintentional

tems to be vulnerable to either attacks over the network, unintentional software malfunctions of programs, or hardware failures in some part of the system. Thus, a process in a multiuser system can terminate due to a failure, even if the failure does not occur while that process is running or if it occurs in a part of the hardware the application never uses. Operating systems are often able to isolate software failures within applications. However, as operating systems get larger and more complex, there are more situations where the operating system itself is affected by a failure and brings down all applications running on it. Partitioning helps isolate the effects of failures to the partition where the failure occurs. Thus an event that causes a crash in a guest operating system will bring down only that operating system and its applications. To restore the system, only the operating systems running on the affected partitions need to be rebooted. Other operating system images running on the machine continue without being affected by the failure.

Faults may occur in either software or hardware. An example of a software fault is a bug in system software that produces a pointer to an invalid memory region. Most partitioning techniques isolate software faults. Hardware faults, on the other hand, may or may not be local to a partition. If there is a hardware fault that causes an arithmetic unit to produce wrong results, the effect of the fault will depend on the nature of partitioning. In partitioning schemes where the same processor is time multiplexed between several partitions, a failure can affect more than just the partition that manifests the fault. On the other hand, when the system is *physically partitioned,* a processor is assigned to no more than one partition, and hence only that partition will be affected by the failure. Physical partitioning, where each partition uses hardware resources distinct from other partitions, is the closest one can get to isolating hardware faults.

Of course, the reliability of the partitioned system also depends on the reliability of the VMM itself. The VMM typically consists of a program, either in software or in microcode, assisted by special hard-

ware. The reliability of the VMM depends on the reliability of both these parts, though the hardware added is typically small and does not affect reliability in any significant way. The software portion either should be verifiably correct or should be kept small and simple. Contemporary software VMMs are generally quite small, often a couple of orders of magnitude less complex than large operating systems.

9.1.2 Mechanisms to Support Partitioning

As shown in **Figure 9.1**, partitioning needs the help of an additional layer between the physical host hardware and the virtual multiprocessors. We continue to refer to this layer as the *virtual machine monitor* (VMM). The functions to be performed by the VMM determine the complexity of this layer and how it is implemented.

As we observed in **Chapter 8**, for many ISAs, there can be an appreciable degradation of performance because of the need to execute a guest operating system in nonprivileged mode. Commercial partitioning techniques on large shared-memory systems often provide hardware features to improve the performance of the VMM. In fact, VMMs may be implemented completely in hardware, in microcode supported by hardware, or in software supported by hardware. Performance is traded off for greater flexibility as more of the features of the VMM are implemented in software.

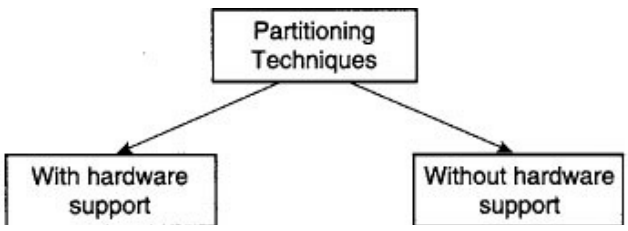
A key difference between techniques that take advantage of hardware modifications and those that do not is the availability of a new privilege mode in which the VMM can operate. This avoids the need to run a guest operating system in user mode and hence the need to take special care of privileged operations executed by a guest OS. One disadvantage of making hardware modifications to support virtualization is the inability to virtualize in a recursive manner—i.e., to run a software VMM on top of the hardware-aided VMM.

9.1.3 Types of Partitioning Techniques

As described earlier, the spectrum of possible implementations for a large multiprocessing system with n processors has two extreme points. At one end is a simple cluster of n nodes, each consisting of a single processor. The natural number of system images (operating systems) in such a system is *n*, each operating system running natively on a dedicated processor node. At the other extreme is an *n*-way shared-memory system with *n* processors, all sharing the same main memory. Here, a single operating system runs across the entire system so that any application on the system can use all the processors while taking advantage of the large shared memory.

Figure 9.3 depicts the various partitioning techniques that have been either implemented or proposed in the literature. As mentioned in the previous section, the implementation of the VMM plays an important role in partitioning, and we have divided the class of partitioning techniques into those that use special hardware for the VMM and those that do not.

On the left-hand side of **Figure 9.3** are techniques that take advantage of special hardware to provide efficient virtual machines. These fall under two categories, *physical partitioning* and *logical partitioning*. In physical partitioning, each image uses resources, processors in particular, that are physically distinct from the resources used by the other operating system images. Thus, the number of partitions that can be supported in such a system is limited to the number of processors, though other considerations may limit this to an even lower number.



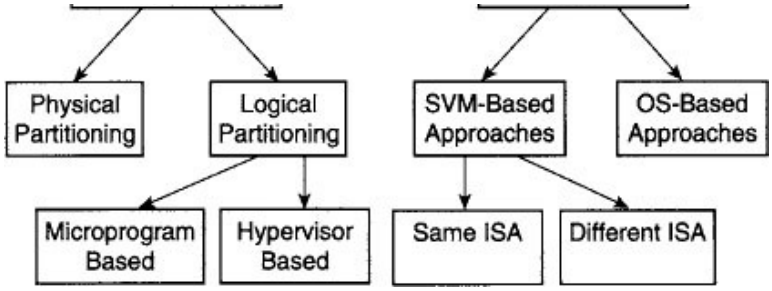


Figure 9.3 Different Types of Partitioning Techniques.

With logical partitioning, images share some of the physical resources, usually in a time-multiplexed manner. Logical partitioning thus makes it possible to partition an *n*-way system into a system with more than *n* images if so desired. Logical partitioning is more flexible and needs additional mechanisms to provide the services needed to share resources in a safe and efficient way. Early logical partitioning techniques implemented the VMM in microcode, i.e., in a programming layer higher than the hardware but hidden from general users of the system (**Borden, Hennessy, and Rymarczyk 1989**). This layer is considered part of the hardware and is sometimes referred to as the *firmware.* Today, the VMM is often implemented as a codesigned firmware-software layer, often referred to as the *hypervisor*. The software component of the hypervisor adds flexibility and provides an interface to system programs that configure the system.

The right-hand side of **Figure 9.3** shows partitioning techniques that do not use any special hardware. We have seen in **Chapter 8** that the VMM of a system virtual machine provides an elegant way to logically isolate multiple operating systems on a single system. This technology can be used to create multiple virtual shared-memory multiprocessors also. Extensions to VMM methods can achieve many of the major desirable features of partitioning, particularly some degree of physical isolation of operating systems. System VM-based approaches may impose some more overhead, compared to the hardware-supported techniques listed in the left half of **Figure 9.3**. However, they are very flexible and, as the figure shows, may

be the only way to provide a virtual multiprocessing system whose ISA is different from the ISA of the native multiprocessing system.

It may be argued that partitioning of resources is really a function that could be provided by an operating system alone, and indeed there are some operating systems that provide ways of partitioning hardware resources among the processes. While this may help isolate processes from one another, it does not provide a virtual "machine" to the user (or group of users) who may wish to run a different operating system.

This chapter will examine many of the partitioning techniques illustrated in **Figure 9.3**. Physical partitioning is described in **Section 9.2**, while two variations of logical partitioning are described in **Sections 9.3** and **9.3.3**. **Section 9.4** describes the system VM-based partitioning, using Stanford University's Cellular Disco system as a case study. Almost all the work done to date on multiprocessor virtualization assumes that the ISA of each processor in the guest system is identical to that of the host system. **Section 9.5** attempts to list the issues involved in virtualizing systems with different guest ISAs and, in particular, systems with different memory model characteristics in the guest.

9.2 Physical Partitioning

Of all forms of partitioning, the physical partitioning approach is perhaps the simplest to understand and the easiest to implement, and it imposes little overhead on an executing application. Different manufacturers allow their systems to be physically partitioned in different ways. While they often use different terminology, the fundamental feature of these various approaches is that they allow a partition to "own" its resources physically, so there is no danger that another partition will accidentally or maliciously compromise the system security or availability.

Unlike some of the other partitioning techniques, control of the configuration of each partition is mostly in hardware. Physical partitioning does not need sophisticated algorithms for scheduling and management of resources. There is typically a central control unit that receives commands from the console of the system administrator and sends out special commands to the various hardware resources, particularly the system boards, configuring them for use in a partition. This may include, for example, information about how the memory on a board maps to the real address space of the partition.

Once a partition is configured, the operating system is loaded. Loading an operating system into a partition accomplishes the functions generally associated with bootstrapping a system; other partitions on the system are unaffected during this operation. Each partition can run an operating system, which may be different from the operating system running on the other partitions. **Figure 9.4** shows a 24-processor system consisting of six physical units (e.g., a board), with four processors on each board, some memory, and I/O logic connected to a set of disks. The six boards are divided into three partitions, with one board dedicated to the first partition, two to the second, and three to the third. The disk units are independently partitioned in a similar way, as shown. The entire system is controlled through a main console (not shown).

As mentioned earlier, several vendors offer large shared-memory systems that can be physically partitioned. Sun Microsystems markets a large SMP server that can be partitioned into several *domains* (Sun 1999). However, each domain must be located in a physical unit that is distinct from that of another domain. A physical unit comprises one system board, consisting of up to four processors, 4GB of memory, and four I/O buses. A partition can span multiple system boards; however, a system board can belong to only one partition—it is not permissible for two partitions to share the resources

tion—It is not permissible for two partitions to share the resources of a single system board.

Hewlett-Packard also allows physical partitioning in their large server systems (**Hewlett-Packard 2000**). These partitions are referred to as *nPartitions.* Like the Sun systems, each partition is constrained to one or more boards, which are also referred to as *cells.* A cell consists of up to four processors connected as an SMP, sharing up to 16GB of memory, and up to 12 PCI slots. HP imposes further restrictions on its partitions—each cell of a partition must be identical to other cells in the partition in terms of the number of processors and amount of memory.

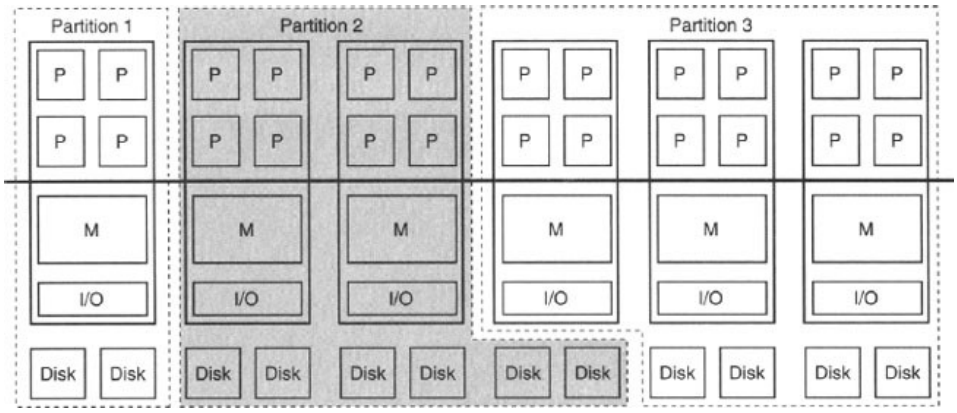


Figure 9.4 Physical Partitioning of a 24-Processor System.

Fujitsu's PrimePower systems (**Fujitsu 2003**), with up to 128 processors, can also be physically partitioned. Each of the system boards contains up to eight processors, memory, and I/O, connected through a crossbar switch. Partitions can span one or more boards. However, PrimePower allows partitions to be smaller than one board. Each board of eight processors can contain up to four physical partitions.

Here are the key advantages of physical partitioning over other forms of partitioning.

- **Failure isolation:** The robustness of a system to faults of various kinds is of considerable importance in server systems. Physical partitioning attempts to

considerable importance in server systems. Physical partitioning attempts to guarantee isolation of a partition from events in other partitions. Physically partitioned systems ensure that in the event of a software failure in a partition, only that part of the physical system that houses the failing partition will be affected. The control unit is designed to be able to reset the partition and reboot the operating system for that partition without the remaining partitions on the system observing any effects of the failure. This isolation also extends to hardware failures. A physical entity such as a board or a multichip module is associated with only one partition. Hence the system can be designed so that the failure of a processor on a board or module need not bring down any of the other partitions on the system. This does not eliminate single points of failure, however. (A single point of failure in a system is defined as a failure that can bring down the entire system.) In most common systems, there are several single points of failure. The control unit, for example, is a single point of failure. If the control unit fails, the partitions of the system are unable to reset themselves and respond to console commands. However, the probability of a hardware fault in a given unit is roughly proportional to the amount of hardware in the unit; and because the complexity of the central control unit is low and its size small, it is less vulnerable as compared to the rest of the hardware. The crossbar switch connecting different processors or different boards in a system is another single point of failure, and major vendors ensure that this switch is built with intrinsically reliable technology, with robust communication paths, and with sufficient redundancy to ensure very high mean time between failures (MTBF).

- **Better security isolation:** Each partition is protected from the possibility of intentional or unintentional denial-of-service attacks by other partitions. Even though each partition can run with its own system administrator, a system administrator on one partition cannot take unauthorized action in some other partition.
- **Better ability to meet system-level objectives:** System-level objectives usually result from contracts between system owners and users of the system. System users pay for specific amounts of computing resources that are guaranteed by the system. Physical partitioning creates partitions more similar to hardware systems than other forms of partitioning. Techniques used to direct resources to applications in stand-alone systems can be applied more readily and predictably in physically partitioned systems as compared to logically partitioned systems.

While physical partitioning has a number of attractive features, it is probably not the ideal solution if system utilization is to be optimized. It is often the case that each of the physical partitions is underutilized, for example, because the system-level objectives force the VMM to allocate resources conservatively. Dynamic workload

balancing is also difficult in physical partitioning because of the physical constraints placed by fault isolation requirements. Logical partitioning sacrifices such physical isolation of partitions in exchange for greater flexibility in allocating resources to partitions.

9.3 Logical Partitioning

Logical partitioning, like physical partitioning, is a way of providing the illusion of several shared-memory systems on a single large shared-memory system. Unlike physical partitioning, however, the partitioning is "logical," in the sense that a partition has no distinct, time-invariant physical boundaries. In logical partitioning the state of two virtual machines may be completely intertwined, not just in memory, but even in the processors. As we noted at the end of the last section, workload balancing is an important goal of logical partitioning, and this is often best accomplished by sharing resources, especially processors, among the multiple partitions in a time-multiplexed manner.

Logical partitioning was introduced on mainframe computers by Amdahl in 1984, IBM in 1988, and Hitachi in 1989. Even though the VM/370 concepts were well established and users had been running several operating systems simultaneously on a single machine, it was felt that an approach was needed that (a) did not require the guest operating system to run in user mode, (b) incurred a lower penalty when running an application on a virtual machine as compared to the native execution, and (c) did not require a complex virtual machine monitor, which, in those days, had almost the same complexity as an operating system. These goals were achieved in Amdahl's multiple domain facility (MDF) (**Doran 1988**), in IBM's logical partitioning (LPAR) (**Borden, Hennessy, and Rymarczyk 1989**) using the processor resource/systems manager (PR/SM) feature, and in Hitachi's multiple logical partition feature (MLPF). All three systems used low-level hardware and microcoded firmware to achieve

this objective. Firmware provides a level of flexibility that allows extensions to be added to a system even after it has been shipped.

From the point of view of the operating system and the applications that run on it, a logical partition behaves in a way similar to a physical partition. In most cases, the version of the operating system designed to run on the native machine can be loaded unchanged on a partition. The characteristics of the partition determine the characteristics of the machine the operating system sees. However, unlike on a physical partition, the hardware resources available to a logical partition may be shared with other partitions. As with other system-level virtual machines, the logical partition only has the illusion of owning the resources that are allocated for its use.

9.3.1 Major Features of Logical Partitioning

Allocations of physical resources are made in logical partitioning during initialization of the partition, as was the case with physical partitioning. The main difference is that allocations must observe certain physical constraints in physical partitioning but not in logical partitioning. This is illustrated in **Figure 9.5**. It may be observed in the figure that even though the sum of the number of processors needed by the three partitions is 28, the partitions can be accommodated in the 24-processor system because four of the processors are shared between partitions 1 and 2.

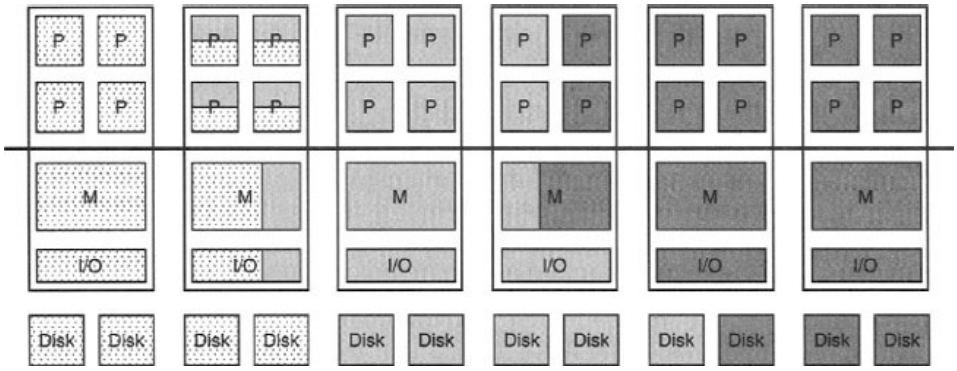


Figure 9.5 A 24-Processor System Divided into Three Partitions. *Partition 1 (shaded with*

dots) needs eight processors but only four dedicated processors; partition 2 (shaded light gray) needs ten processors, with six of them dedicated; partition 3 (shaded dark gray) needs ten dedicated processors.

Some of the major features of logical partitioning using firmware follow.

- **Allocation of processors:** Either a partition may specify a total amount of processing power it needs and leave the allocation of available processors to workload management software, or it may specify that specific processors in the system be dedicated for its use. A partition can specify that it needs a certain number of processors but that it is willing to share these processors with other partitions. For example, if a total of four processing units is needed by the partition, the partition can specify either that it needs four processors dedicated to itself or that it needs eight processors but only half the available compute power in each of the processors.
- **Allocation of memory:** The total shared-memory is allocated to partitions in chunks of large granularity, e.g., 1MB. Smaller chunks increase the bookkeeping to be performed by the firmware monitor, while larger chunks restrict the flexibility of partitioning. The conversion of real addresses in the address space of a partition to a physical address in the larger memory is done in hardware.
- **Allocation of I/O resources:** A partition gets its own I/O subsystem. An I/O device that has multiple ports may be shared among various partitions that are each allocated a subset of the ports. In the simpler partitioning schemes, ports are allocated when the partition is initialized.
- **Communication between partitions:** In order to allow easy migration of cluster systems to partitioned systems, the modes of communication commonly implemented on clusters are also supported between partitions. Typically such communication uses either shared storage devices or network commands. A partition looks like a separate machine to the rest of the world, including to other partitions.

In the next section we provide a more detailed description of the IBM LPAR system, which implements logical partitioning on mainframe computer systems.

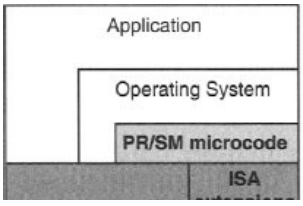
9.3.2 Case Study: The IBM System/390 Logical Partitioning Feature

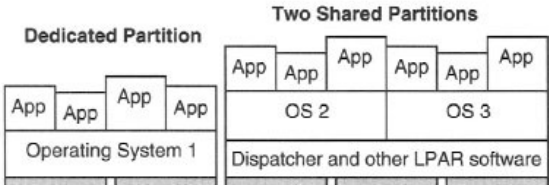
IBM mainframes provide a processor resource/systems manager (PR/SM) that allows a S/390 machine to run multiple operating s

(PR/SM) that allows an SMP complex to run multiple operating system images using logical partitioning (**Borden, Hennessy, and Rymarczyk 1989**). The PR/SM consists of special hardware and microcode that can be invoked and controlled directly by the system administrator through the machine console. The machine mode that permits exploitation of the PR/SM feature is termed LPAR (logical partitioning). In this mode, the system can have several partitions—up to six on the ES/3090S systems, where it was first introduced.

Each logical partition is essentially a collection of hardware resources, including processors, memory, and I/O, that are needed to support an operating system. Each logical partition can support an operating system different from those on other partitions. Besides an operating system, a partition can also run the VMM of a conventional system virtual machine. Partitions are logically independent of other partitions—they communicate with each other in ways similar to nodes within a cluster. These methods of communication include shared storage devices, channel-to-channel communication, and network commands.

Some of the hardware resources of the machine are divided between the logical partitions. Each partition sees a portion of the physical memory on the system. The I/O elements, such as the channel paths, subchannels, and logical control units, are also divided among the partitions. On the other hand, any processor in the system can be either dedicated to a partition or shared among multiple partitions. Each partition can use one or more logical (or virtual) processors. Overcommitment of resources is permitted—the total number of processors used by all the partitions together can exceed the actually available number of physical processors.





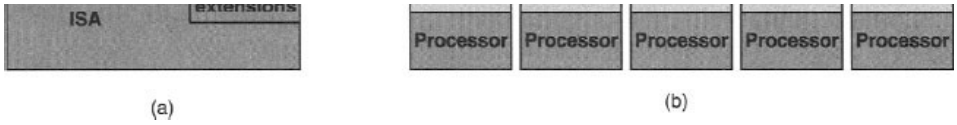


Figure 9.6 The LPAR Mechanism. *(a) A snapshot in time of the various active layers on the system. The base hardware has additional instructions and microcode as compared to a standard processor. This hardware enables switching between different partitions. (b) A system view of LPAR. The physical processors may each be either dedicated to a single partition or shared among multiple partitions of potentially different sizes. Allocation of logical processors of a partition to physical processors is done dynamically using dispatchers and job schedulers in the LPAR software.*

Contiguous regions of physical main memory are allocated to each partition at 1MB granularity. All references to real memory in a partition are remapped to the appropriate region of physical memory. A check is made to ensure that references are within the range specified for the partition.

Each partition is given its own logical I/O subsystem. A given device is associated with multiple channel paths to allow the device to be shared among different applications. These channel paths are divided among the various partitions and hence provide the desired isolation.

The number of processors used by a single partition cannot exceed the number of physical processors available on the system. The user of a partition can determine whether a partition should be dedicated or shared. A dedicated partition has exclusive use of the physical processors allocated to it and is probably the most appropriate choice for a partition that has a steady demand for the computing resources it uses. Most workloads tend to have demand variations from a peak at certain times to light use at others; in these cases it may be more cost effective to use a shared partition and allow the sharing of processors with other partitions on the system, as illustrated in **Figure 9.6**. Each processor allocated to a shared partition is associated with a user-defined *weight* that is an indicator of the partition's priority for use of that processor in relation to other partitions that also use the processor.

Partitions on the IBM System/390 are defined by the user, who speci-

Partitions on the IBM System/390 are defined by the user, who specifies

- The name of the partition
- The I/O configuration
- The memory configuration
- The processor configuration

Each machine has an operator console used to perform a range of functions, at three different levels of the system hierarchy: to manage either the physical configuration of the entire system, the configuration of the LPAR environment, including the resources within each partition, and the configuration of the operating system running within a partition.

The LPAR functions are largely controlled through the system console. The resources are not dedicated to a partition until the partition is activated and it has been determined that the partition has all the resources it needs for its execution. When all the resources it needs become available, a ready partition is activated. For example, a shared partition is activated only when the number of processors it needs is no more than the total number of physical processors minus the number of dedicated processors allocated to active partitions. Activation is the logical equivalent of the power-on reset performed to boot up a system. The system then goes through the sequence of loading the desired operating system on the partition.

While most of the resources allocated to a partition remain with the partition, there are occasions when it is necessary to move resources between partitions. For example, there may be an I/O device, such as a special recording unit, that is expensive. Only one such device may be supported on the system, but the unit may never be used for long periods of time. In this case, the system administrator can remove the device from one partition and allocate it to another partition that requests it. Reconfiguration of channel paths from one partition to another on the System/390 is also done through the system console.

When a processor is shared between partitions, it is desirable to allocate the processor resource in a fair manner, consistent with the weight of each of the partitions. Scheduling logical processors to run on physical processors is done by a software program called the LPAR workload manager, which runs in a partition of its own. It makes scheduling decisions based on the expected response to I/O operations and on the relative utilization of the available processors. The LPAR dispatcher maintains the state of each of the active partitions and loads the hardware registers with the appropriate contents when it schedules one of the active partitions to run on the processor. Here are some of the considerations for dispatching a waiting partition.

- **The weight assigned to the partition:** The priority of a waiting partition is determined by the weight of a partition relative to the weights of the other waiting partitions.
- **The activity within a partition:** When a logical processor in a partition enters a wait state (i.e., the state in which it is waiting for some external event, such as I/O, to occur, as opposed to waiting to be scheduled), it is usually more productive to unload the partition and load another partition to execute on the processor.
- **High-priority I/O interrupt:** When an I/O interrupt is received for a logical processor that is waiting for the interrupt, the logical processor replaces the logical processor of another partition that is currently executing, provided the priority of the waiting partition is higher than the priority of the current partition.
- **Interval timeout:** The dispatcher allows any partition to execute for a predetermined maximum time. If the logical processor is still active at the end of this dispatcher timer interval, it is swapped out of the system. This is a useful mechanism to prevent either runaway partitions or malicious partitions that could otherwise monopolize the system and prevent forward progress for the other partitions on the system.
- **OS-initiated swap:** It could happen that the logical processor of a partition chosen via the foregoing criteria is in an unproductive mode, e.g., an idle loop. There is a mechanism provided on the System/390 that allows an operating system or a VMM running on a partition to inform the dispatcher that it is in an unproductive mode and so may be swapped out.

IBM's LPAR provides fault containment as expected from a logical partitioning scheme. Any software fault that occurs can cause the

partitioning scheme. Any software fault that occurs can cause the failure of only the partition that caused the error. Other partitions cannot be affected by such a fault. Whenever possible, the effect of hardware faults is also limited to the partition that was executing when the fault occurred. A detected fault causes an exception that is transmitted to the partition that was executing. The operating system in this partition then attempts to recover from the fault exactly the way it would have attempted on a native machine. The effect of a hardware fault is localized to a single partition, especially in the case of dedicated partitions. However, unlike physical partitioning, there still are several cases where the failure of a shared component could bring down several partitions or even the whole system.

Today, there are few systems, other than IBM's mainframes, that offer logical partitioning through microcode. With the advent of RISC instruction sets and the resulting move away from firmware implementations, it is more common to find systems that implement the microcode functions in a software layer similar to codesigned virtual machines. This software layer, called the *hypervisor*, is the subject of the next section.

9.3.3 Logical Partitioning with Hypervisors

Logical partitioning has been introduced on several systems that do not have microcoded processor implementations. Examples include the current IBM's iSeries servers based on the AS/400 (**Boutcher 2001**), HP-Compaq's Superdome servers (HP 2002), and IBM's pSeries AIX servers. The complex microcoded instructions that support logical partitioning on the IBM System/390 are replaced by programs that use the basic ISA of the host platform and run in a special mode that is more privileged than all other software on the system. Thus the definition of a new mode of operation is what distinguishes this class of partitioning. This new mode is used by the hardware vendors to provide partitioning capability. If the mode is not exposed in the ISA, then the software that runs in this mode can be

viewed essentially as an extension of the hardware itself, very much like the VMM software in a codesigned virtual machine. The common name given to this piece of software is the *hypervisor*.

In order to limit the vulnerability of the system to software failures, an important characteristic of any hypervisor is its small size. Like operating systems in nonpartitioned systems, hypervisors have ultimate control of all resources on a machine. Unlike operating systems though, hypervisors tend to be unobtrusive—their main function is to configure the system and then get out of the way, allowing the hardware that has been allocated to the partition to work directly with the operating system in the partition.

9.3.4 Comparison with System Virtual Machines

Hypervisors and conventional system VMMs are similar, in that both run in the highest-privilege mode. However, the principal difference between the two is that hypervisors need hardware support and work in a special mode, while system virtual machines may be implemented on standard unmodified hardware. The guest operating system in a logically partitioned system works in the privileged mode, just as it would on native hardware, whereas in a conventional system VM, the guest operating system works in user mode. Applications running on partitioned systems tend to work more or less at native speed, especially with dedicated partitions, which have exclusive use of the processors allocated to them.

Today's system VMs on the IBM zSeries (System/370 descendents) can support very large numbers of virtual machines, often several times the number of processors on the system. In comparison, because of the high-performance requirements and the fault-tolerance requirements of the markets to which they are targeted, partitioned systems restrict the number of partitions on a system to much smaller numbers. A 64-way HP Superdome system, for example,

smaller numbers. A 64-way IA-32 supercomputer system, for example, supports no more than 64 partitions.

Whether logical partitioning is better than a conventional software-based VM is likely to be debated for a long time. There is a certain elegance to the software VM approach that aims to provide a virtual system on unmodified hardware, i.e., on any implementation of the original, untouched architecture. Also, adding special hardware features to allow the coexistence of multiple operating systems on the machine virtually eliminates the possibility of running the virtual machine monitor on a copy of itself. However, at least within IBM, the software only nature of the VM approach had already been compromised soon after its introduction by the incorporation of special performance-enhancing hardware assists for VM. In fact, when IBM designed the PR/SM hardware/microcode feature, it not only included in the design special mechanisms to support logical partitioning, but also leveraged these mechanisms to further improve performance for conventional system VMs.

9.3.5 Hardware Support for Logical Partitions

Even if most of the monitor functions can be incorporated into the hypervisor software, there are hardware implementation changes that need to be made to support logical partitioning.

Hypervisor state registers: The hypervisor state may have its own set of registers that are not accessible to software in any other mode. For example, as we will see shortly, there are special registers that isolate the real memory of one partition from the real memory of other partitions and also from the real memory used by the hypervisor itself.

Replicated registers: Some of the common architected registers may be replicated, depending on how often the hypervisor is invoked. Amdahl's MDF (**Doran 1988**), for example, replicated the general-purpose registers and all other registers except the floating point