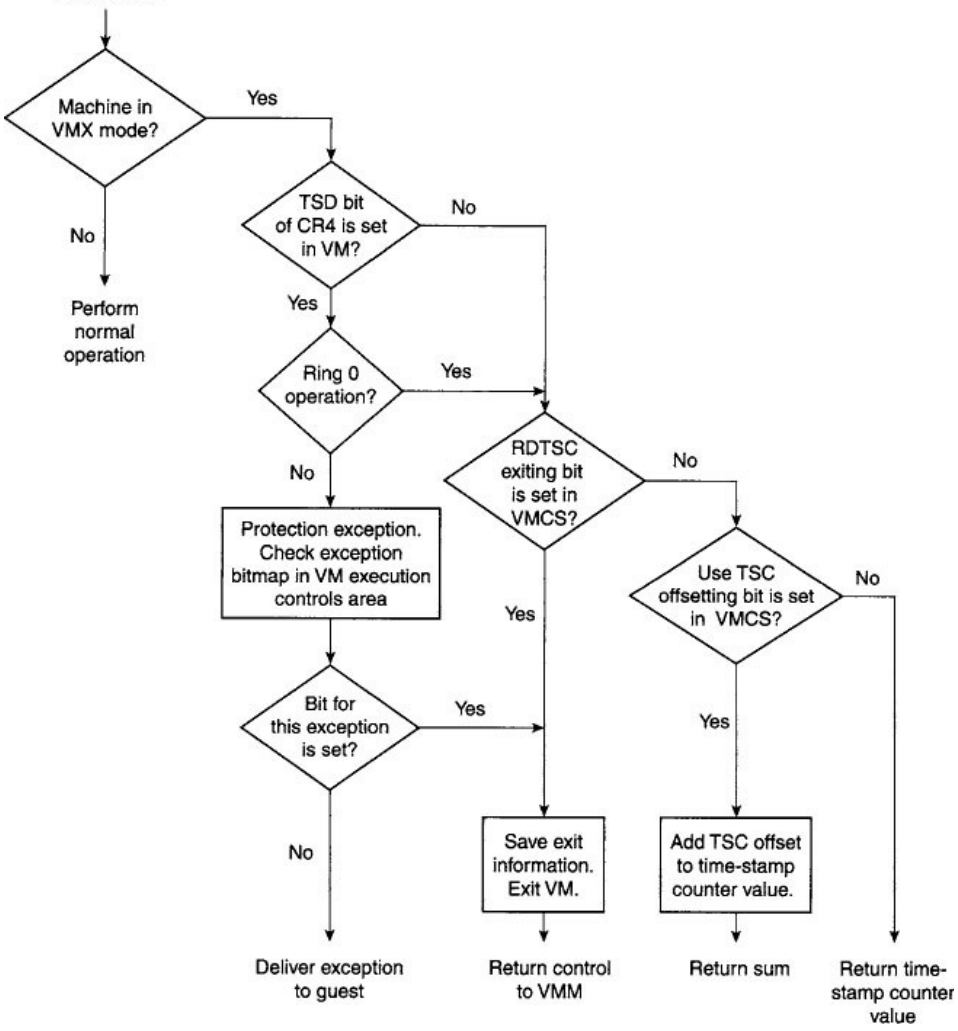


Figure 8.27 Actions Taken by Hardware When a Read Time-Stamp Counter (rdtsc) Instruction Is Encountered.

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**.

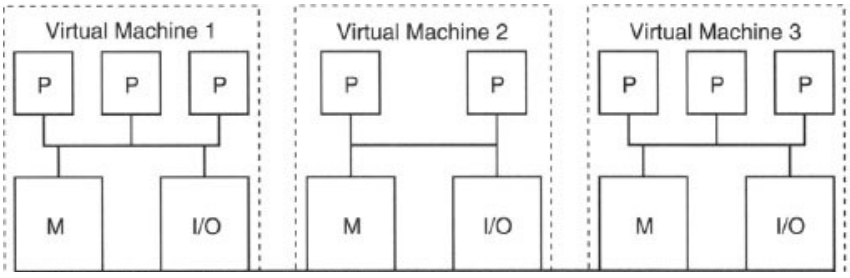
9.1.1 Motivation

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.



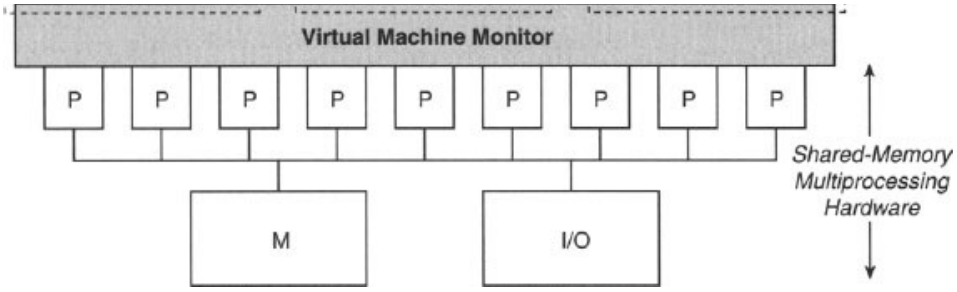
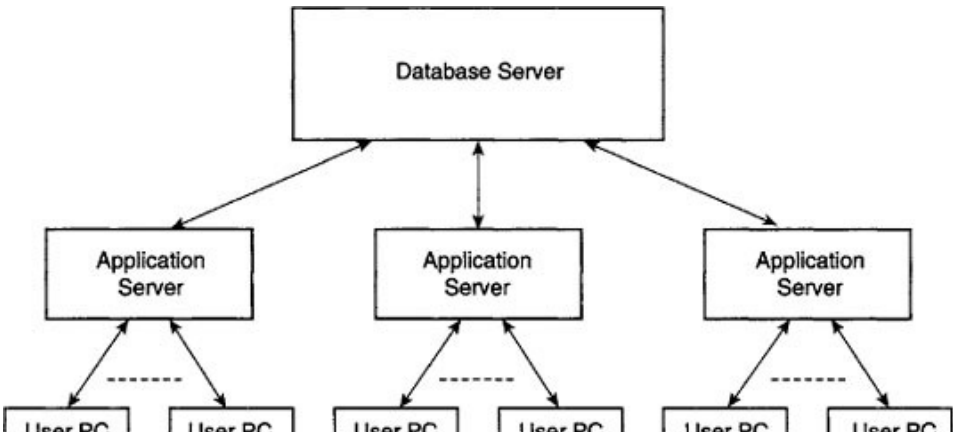


Figure 9.1 Large Shared-Memory System Partitioned into Smaller Shared-Memory Systems.

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.



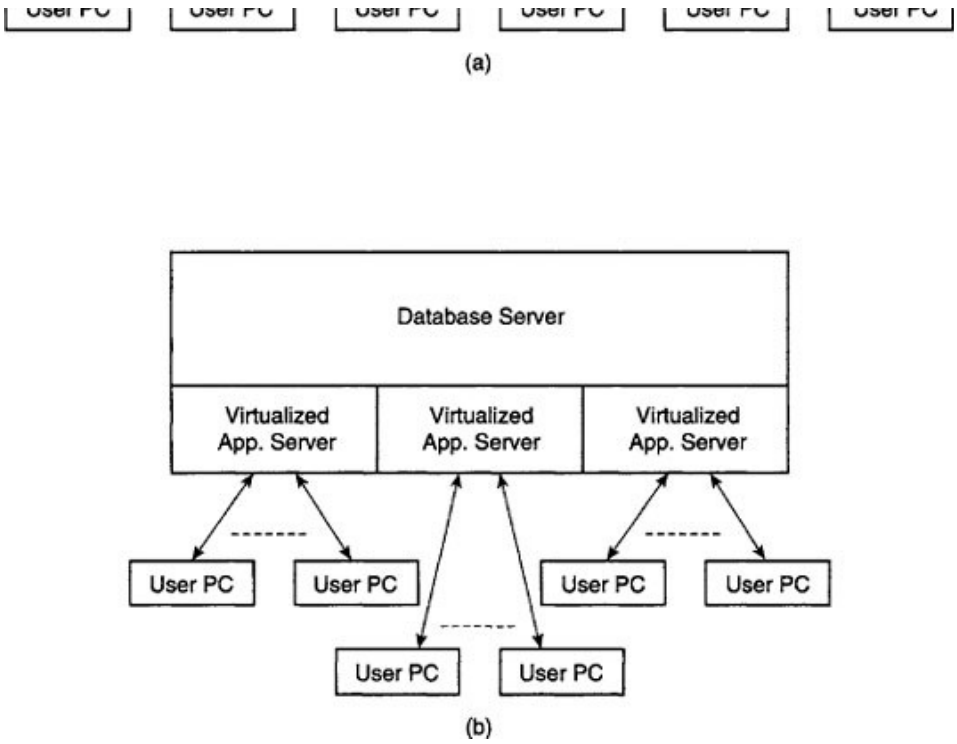


Figure 9.2 Consolidation of Database and Application Servers on a Single Platform. (*a) Typical three-tier server model. (b) Consolidation of the application server with the database server.*

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.

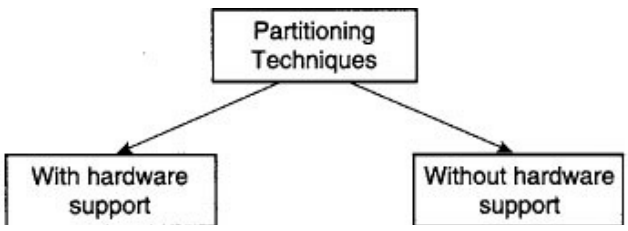
A key difference between techniques that take advantage of hardware modifications and those that do not is the availability of a new privilege mode in which the VMM can operate. This avoids the need to run a guest operating system in user mode and hence the need to take special care of privileged operations 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.



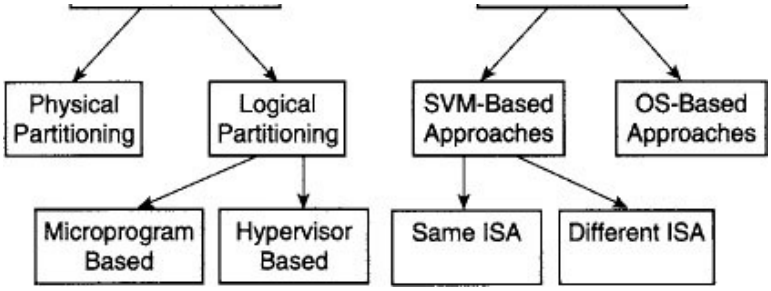


Figure 9.3 Different Types of Partitioning Techniques.

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.

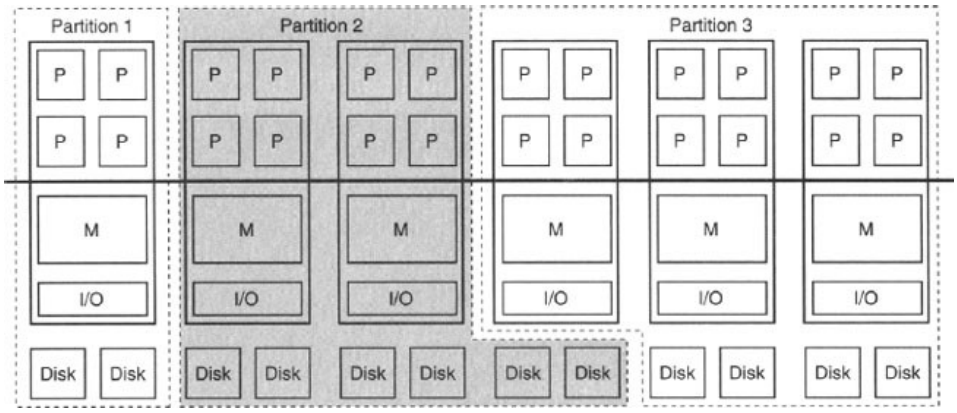


Figure 9.4 Physical Partitioning of a 24-Processor System.

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.

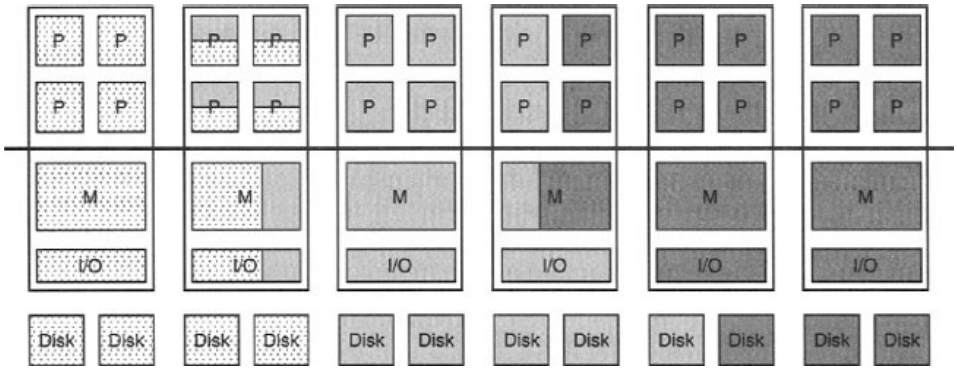


Figure 9.5 A 24-Processor System Divided into Three Partitions. *Partition 1 (shaded with*

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.

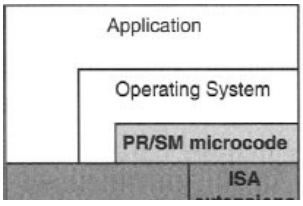
9.3.2 Case Study: The IBM System/390 Logical Partitioning Feature

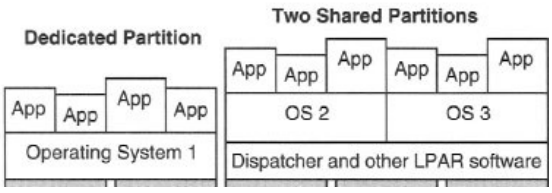
IBM mainframes provide a processor resource/systems manager (PR/SM) that allows an SMP complex to run multiple operating sys-

(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.





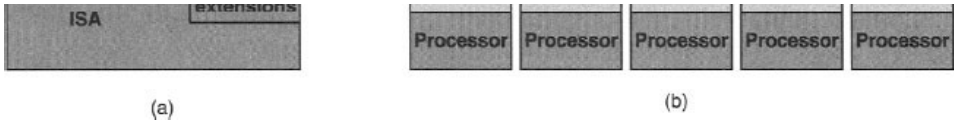


Figure 9.6 The LPAR Mechanism. *(a) A snapshot in time of the various active layers on the system. The base hardware has additional instructions and microcode as compared to a standard processor. This hardware enables switching between different partitions. (b) A system view of LPAR. The physical processors 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 Superdome 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

purpose registers and all other registers except the floating-point registers, which were not used in the hypervisor mode. Registers that are replicated need not be saved on a mode switch and do not occupy space in memory—this was an important consideration when real memory was much smaller than it is today. Another reason for the replication was the extra level of protection provided by physical separation of the registers, especially against bugs in hypervisor software.

Hypervisor memory: The hypervisor needs to access memory, especially to save information about the various partitions as well as to save the relevant state of each partition on a mode switch. Thus, the physical memory of a system, which in nonpartitioned systems is owned by the operating system, must be under the control of the hypervisor in a logically partitioned system.

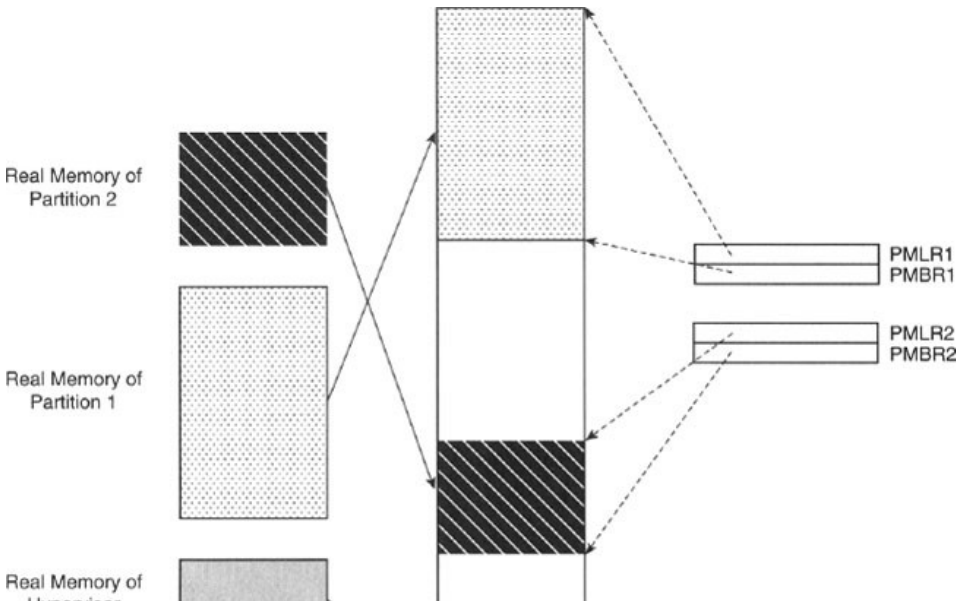
One way to do this is to partition the physical memory into disjoint regions. One of these regions is allocated to the hypervisor itself, while each of the other regions is allocated to each of the logical partitions currently active on the system, as shown in **Figure 9.7**. Corresponding to each partition is a *partition memory base register* (PMBR) and a *partition memory limit register* (PMLR), which determine the bounds of the physical memory accessible only to the partition.

If the hypervisor is allocated an appropriately sized contiguous chunk of the low address space of physical memory, it is possible to bypass translation for memory accesses in the hypervisor mode—the effective address computed in an instruction becomes identical to the physical address. Addresses in a partition, however, must be relocated.

The virtual address generated by a partition is converted to a real address via normal address translation mechanisms (see Section A.3.4). This real address must then be added to the value contained

in the PMBR of the current partition to determine the actual physical location of the address. In order to prevent a partition from accessing the physical memory assigned to another partition, the physical address generated must also be checked against the PMLR. An attempt to access a location beyond that indicated in the PMLR corresponds to an attempt by the partition to access a real-memory location beyond the size of the real memory expected by the operating system in the partition and should generate a real-memory violation exception.

The conversion of a partition real address to a physical address could be done in hardware by adding one more stage to the address translation process. However, this penalty can be avoided in a processor with a hardware TLB by simply replacing the real-memory address field in the TLB entry with the physical address. This, of course, assumes that the size of real memory allocated to a partition is an integral number of pages and that it is smaller than the size of the actual physical memory available. In fact the sum of the sizes of real memory of all partitions on the system must be smaller than the size of physical memory, leaving some of the physical address space to the hypervisor. This suggests the reason that typical logically partitioned systems allow only a few logical partitions.



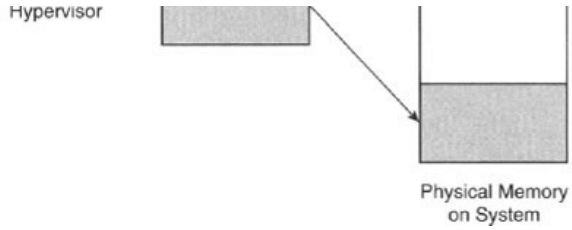
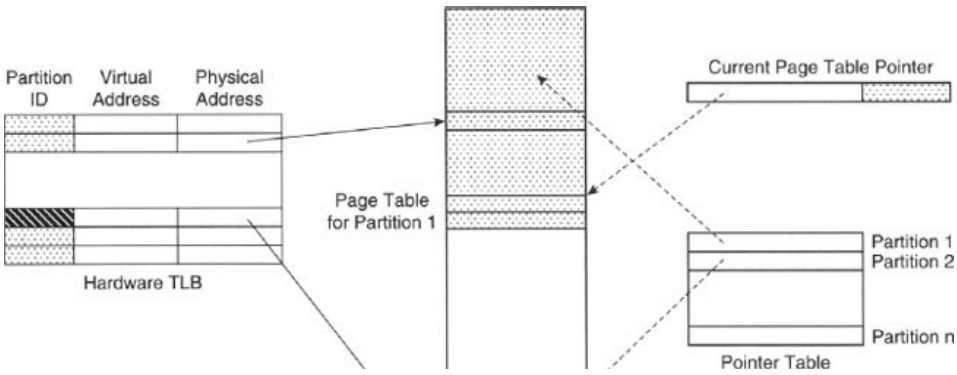


Figure 9.7 Mapping of Real Memory of Partitions to Physically Available Memory. *When contiguous chunks of physical memory are allocated to each partition, the mapping table is simple and needs just two pointers as shown.*

Any TLB miss handling depends on whether the system implements a hardware- or software-managed TLB. In a hardware-managed TLB, the miss is handled by the hardware walking the architected page tables to find the page mapping. When the real address corresponding to a virtual address is found by the hardware, one of the TLB entries is replaced and overwritten as usual with the virtual and real addresses of the new entry. But in addition, the PMBR of the partition is used to establish the physical address of the page in the TLB entry.

Efficient and secure mapping to physical addresses requires that the page table, which contains information about physical addresses, be invisible to the operating system in each partition. Thus modifications to the page table must be done in hypervisor mode. For efficient servicing of page faults, each entry in the TLB is tagged with the identity of the partition to which that entry belongs. As shown in **Figure 9.8**, the page table pointer is under the control of the hypervisor and must be reloaded with the appropriate pointer whenever one partition is swapped for another.



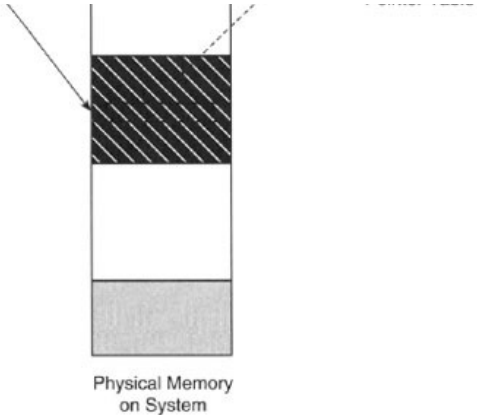


Figure 9.8 Adaptation of a Hardware TLB for Logical Partitioning. *The pointer table is typically small and can be maintained in hardware. The current page table pointer is loaded with the entry in the pointer table corresponding to the currently active partition.*

In the case where the processor implements a software-managed TLB, each partition expects to see a TLB, and hence the TLB for each partition must be virtualized. If the overhead is acceptable, the virtual TLB can be swapped in and out of the physical TLB as active partitions change. Otherwise, the physical TLB can act as a cache for all the virtual TLBs for a processor, with the hypervisor intercepting all actions of a partition that access the TLB and making the appropriate replacement decisions.

Interrupt Handling

There are three kinds of interrupts meant for the hypervisor. These are generated as a result of actions initiated by the hypervisor itself. One kind is an interrupt from an I/O device responding to a query about its operational status. The second kind involves interrupts from the system management console through which a system administrator instructs the hypervisor, for example, about the requirements of a new partition. The third kind involves interrupts from a partition requesting some service from the hypervisor.

In the case of a machine check interrupt, for example, due to an uncorrectable hardware fault, the hypervisor first attempts to determine whether the machine check affects the hypervisor and its tables. If so, there is not much that the hypervisor can do. It may be

able to examine the contents of the tables to determine the partitions affected and reboot these partitions. On the other hand, if the hypervisor itself has not been affected due to the machine check, the hypervisor emulates a machine check for the partition that was running at the time of the interrupt and jumps to a special routine registered by the guest operating system.

This is a general technique that can be used for hardware interrupts meant for a guest operating system. As in the case of the VMM of a system virtual machine, the hypervisor is the first point at which interrupts are handled. The hypervisor determines whether the interrupt is meant for it or for one of the partitions on the system. If the interrupt is for a partition that is not the one currently executing, it may be queued for a later time, e.g., immediately after the partition is reactivated. In special cases, that partition can be reactivated immediately to enable it to handle the interrupt. After an interrupt is handled, control can remain with the partition that handled the interrupt or it can be transferred back to the partition that was executing at the time of the interrupt.

The case of software interrupts (or traps) is considerably easier. The trap can be handled by the guest operating system in privileged mode without the hypervisor's entering the picture.

9.3.6 Hypervisor Services interface

We have assumed thus far that an operating system that works on a native system should work unchanged in a partitioned system. Hardware support for the hypervisor can go a long way in improving the performance of such a system. But it should be clear that the performance can be made even better if a guest operating system is made aware of the fact that it is operating in a hypervisor environment. It may be recalled that, in discussing VM assists in **Chapter 8**, we also noted that the performance of a virtual machine system can be enhanced by a "handshaking" mechanism between the VMM and

be enhanced by a "handshaking" mechanism between the VMM and the guest operating system.

The communication between the hypervisor and the guest OS is carried out using calls from the OS to the hypervisor. Thus, beyond the ISA of the virtual machine is another interface, called the *hypervisor services interface.* A typical call to this interface involves a request for management of a resource that would normally have been under the control of the OS but that would be managed better if handled by the VMM. An example is the hardware page table management on the IBM PowerPC (**Engebretson, Corrigan, and Bergner 2001**).

The PowerPC has an architected page table with an inverted page table structure. When a TLB miss occurs, the hardware accesses the inverted page table, also referred to as the *hardware page table* (HPT) of the processor. When running in the native mode, if no entry is found in the HPT, a page fault interrupt is delivered to the operating system, which determines an appropriate page for replacement and then creates a new entry in the HPT. The situation is slightly different in the partitioned case. The guest operating system now has its real addresses mapped to physical addresses, and hence the mapping in the HPT indicates the virtual-to-physical mapping of the pages in memory. For security reasons, this map must not be accessible to the guest operating system. Instead, it is the hypervisor that has sole control of this table. On a page fault, the guest OS does not directly modify the HPT. Rather, it invokes the hypervisor, using calls from the hypervisor systems interface to establish an entry in the table.

Figure 9.9 shows some of the calls in the hypervisor interface related to page table management in the IBM PowerPC.

Call	Action
`find_valid()`	Find any valid entry for a given virtual address

	given virtual address
Invalidate()	Invalidate a specified entry
add_validate()	Add a new valid entry
modify_pp()	Modify protection status for a specified entry

Figure 9.9 Some Calls in the Hypervisor Interface to Support Page Table Management on the IBM PowerPC iSeries.

It is certainly possible to allow the hypervisor to take over the entire page table management, rather than just provide the services to aid the operating system in this function. However, there are advantages if the hypervisor implements just mechanisms rather than both mechanisms and policies. Policies are often best implemented by the operating system, which has a better view of the requirements of the partition and its applications. Further, restricting the hypervisor to the mechanisms makes it compact and less error prone.

As we had indicated in the case of system VMs, the disadvantage of the hypervisor services interface philosophy is that the operating system is no longer portable. Calls made to the hypervisor may need to be changed when the operating system is ported to another platform. This could be a problem, even with a new implementation of the hypervisor on the same platform—new versions should be designed with backward compatibility of operating systems in mind.

9.3.7 Dynamic Partitioning

In order to provide versatile operation, partitioning techniques must allow for the possibility of changes in the configuration of a running system. Two types of changes can occur.

1. A partition may complete its task, and a new partition may be ready for installation on the system.
2. A partition may find its own needs changing and therefore desire to change its configuration, for example, the number of processors, the amount of memory, or the devices that are attached.

The first type of change is already accommodated on most logical partitioning systems. In order to shut down a partition, the hypervisor must first recover all the resources owned by the partition that is shutting down. For security purposes, it may need to erase (or otherwise make unreadable) information that may be contained in the memory and on the disks being vacated by the partition. It may also need to perform other tests to ensure that the operation of all freed hardware will be identical to that needed by a system that is being booted up. The freed resources will be placed in a pool of resources from which a new partition gets its allocation.

When the request comes to configure a new partition, the hypervisor examines the resource requirements of the partition and determines whether they can be completely satisfied from the pool of available resources. If the request cannot be satisfied, for example, because of insufficient processors or memory, the hypervisor returns a failed-request message, also possibly indicating the reasons for the failure. If the request can be satisfied, the desired resources are removed from the free pool and a new partition is created. At this point the partition can be loaded with the desired guest operating system.

Note that this technique can be used to migrate a partition from its current form to a new form. The need for such a migration may come in two ways. First, an unrecoverable hardware failure may have occurred in one of its components. In this case, the hypervisor can attempt to configure an identical partition using resources from the free pool and then migrate the state of the partition to the new partition before shutting down the old partition.

The need to migrate partitions can also come from unexpected hardware demands of some partition. If, for example, the paging requirements of a partition have grown to a point where its performance is severely degraded, it may be possible to create a new partition with an identical configuration, except with a larger memory,

and migrate the state of the old partition to the new. Of course, such a migration is useful only if the operating system on the partition can also accommodate changes to its memory size in a dynamic manner.

In **Section 9.3.2** we saw how reconfiguration of I/O resources of a partition could be accomplished in a dynamic manner on the IBM System/390 LPAR. A similar technique can be used to increase the allocation of processor resources of a partition. However, reallocation of memory poses a problem. The problem is similar to that faced during dynamic memory allocation—the requirement that the real memory of a partition be mapped as a contiguous physical chunk must be relaxed. Thus, support must be provided in the hypervisor to stitch together several pieces of physical memory to create the illusion of a contiguous real memory.

The way segmentation and paging helps with the problem of dynamic memory allocation on a processor is discussed in Appendix Section A.3.4. Similar schemes can be adopted for memory allocation in partitions. In order to take advantage of paging hardware already available in most systems, allocation of memory between partitions should be made at page granularity. This is feasible, except the failure of a memory module can affect a large number of partitions. Moreover, the number of physical pages on typical systems will be so large that the hypervisor may need a lot of space simply for bookkeeping. Whenever a partition is shut down, for example, the hypervisor would need to go through a large number of entries to ensure that every page belonging to the partition is invalidated and then freed. This problem can be alleviated by choosing a granularity of allocation that is much larger than a page though still considerably smaller than the size of typical main memory allocations, for example, 256MB.



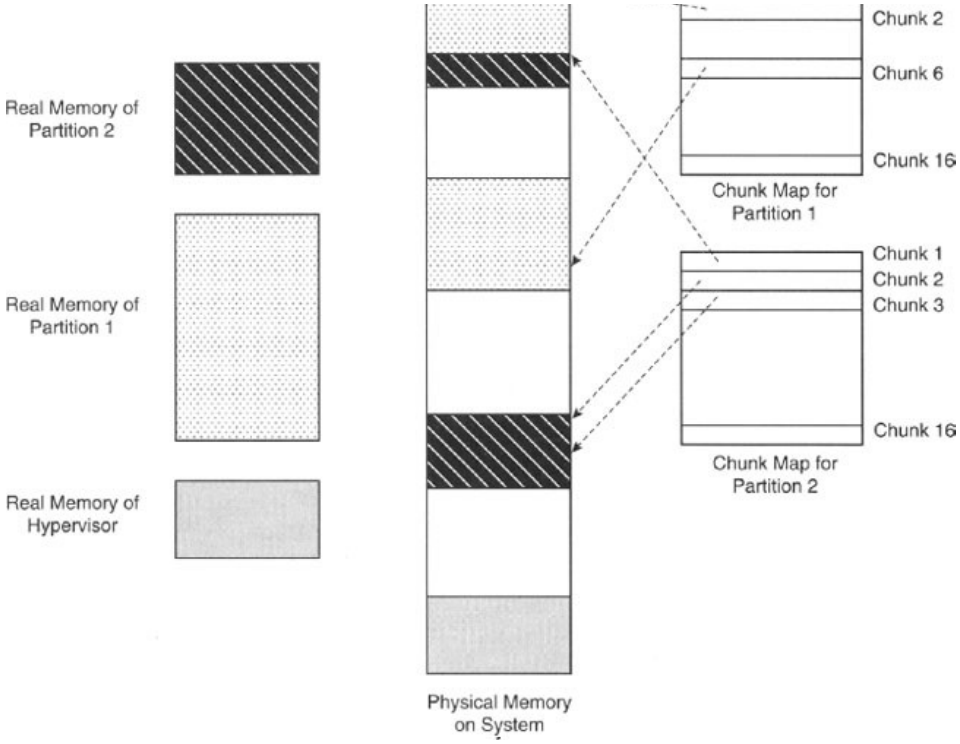


Figure 9.10 Illustration of the Dynamic Allocation of Real Address Space for Partitions Using Chunks.

The hardware implication of allowing dynamic expansion, and perhaps even dynamic contraction, of the memory space of a partition is that it is no longer sufficient to have a single PMBR–PMLR pair associated with each partition. Instead, the hypervisor must keep a map between different regions of the real memory and the physical memory for each partition. For example, by allocating 16 pointers for each partition, memory sizes of up to 4GB can be supported, assuming 256MB chunks, as shown in **Figure 9.10**.

9.3.8 Dynamic LPAR

IBM's PowerPC pSeries servers (**Jann, Browning, and Burugula 2003**) introduced the capability of dynamically migrating resources from one partition to another without requiring a reboot of the affected partitions. This is referred to as *dynamic logical partitioning,* or DLPAR. Resources moved by DLPAR have the same set of capabilities they would have had if they were assigned to the partitions at boot time. Movement of resources currently requires the following

boot time. Movement of resources currently requires the following steps.

1. A request is made to the guest AIX operating system on a partition, asking it to release a resource and, if necessary, to put it in a quiescent state.
2. The AIX operating system stops the resource and releases it to the hypervisor, which places it in its free pool of resources.
3. A request is made to the hypervisor, again through the system console, to allocate the resource to another specific partition that needs it.
4. When successful, a request is made to the guest AIX operating system on this partition asking it to acquire the resource and configure it for its use.

The AIX operating system kernel was modified to run entirely in the virtual mode in order to accomplish DLPAR. In fact, the granularity of memory allocation in IBM's system is a page. Removal of processors had already been implemented in AIX to support graceful degradation in the event of processor failures. Addition of processors was a feature added to AIX to support DLPAR.

9.3.9 Expanding the Role of the Hypervisor

The hypervisor is essentially an extension of the hardware platform provided to the system. The services provided by the hypervisor can therefore include functions that control hardware features of the system.

- The hypervisor can monitor power usage in various parts of the system and feed this information back to the partitions. Operating systems running on the partitions can react to high usage of power and take corrective measures by dropping requirements for the particular resource showing high power usage.
- The hypervisor can provide similar functions in the event of hardware problems. If an error is detected during computation and if repeated attempts do not lead to the elimination of the error, the hypervisor can take measures to move all partitions using the faulty processor to a spare processor and isolate the failing processor and even to replace it with a good processor without halting the rest of the system.

The traditional tasks of the hypervisor are passive in nature—the hypervisor gets commands from the system administrator and either satisfies the request or informs the administrator it is unable to

satisfy the request. This is largely due to the evolution of the hypervisor as an extension of the hardware and hence a desire to make the hypervisor compact, verifiable, and error free. It is conceivable that in the future, the hypervisor will take on the role of a powerful software layer in the hierarchy of layers that make up a system. When this happens, the hypervisor will implement not only the mechanisms but also policies to configure and manage the resources in the system. The hypervisor could then make decisions on its own, based on several different factors, concerning the following.

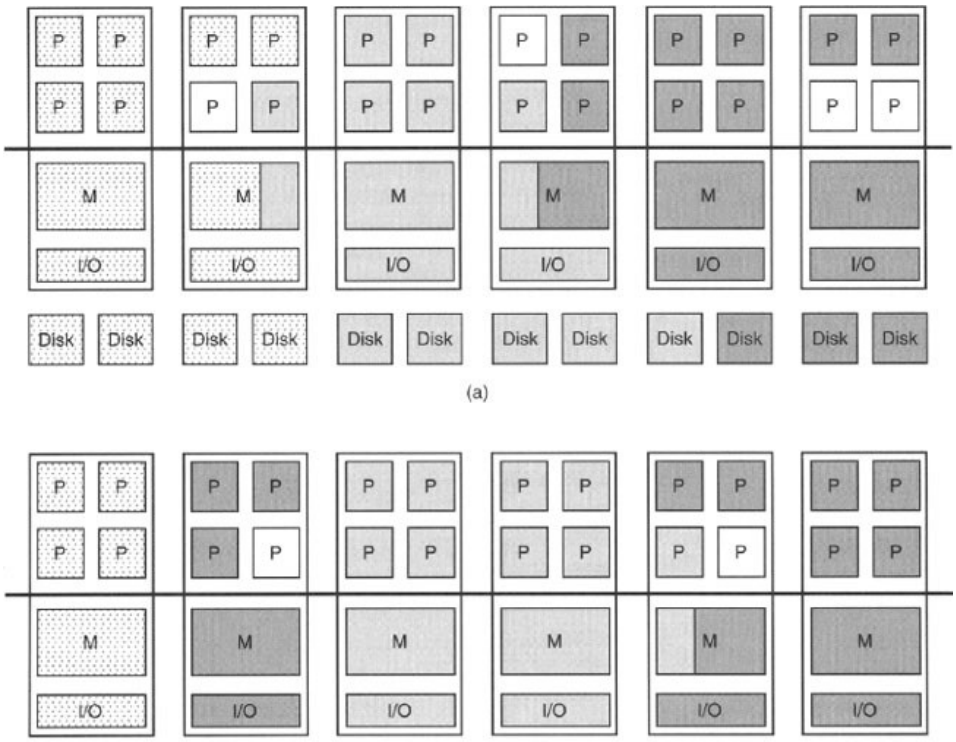
- Environmental factors, such as the power usage, temperature, or reliability of the hardware
- Business factors, such as the security of the system and the fulfillment of various system-level objectives contracted with users of the system
- Operational factors, such as efficiency of utilization of hardware
- Cost factors, such as the revenue generated by the system as a whole

Figure 9.11 illustrates a time sequence where processors and other resources migrate in a dynamically partitioned system to achieve system load balancing.

Indeed, as the complexity of large computing systems increases, it will be increasingly difficult for system administrators, let alone users, to manage resources of the system efficiently and securely. Increasingly, therefore, systems will provide more automation in the areas of resource allocation and reconfiguration. Systems will monitor their own behavior, learn, and adapt to changing situations without intervention from the operator. It is possible that the hypervisor could end up playing a central role in this “autonomic” system of the future (**IBM 2001**).

9.4 Case Study: Cellular Disco System Virtual Machine-Based Partitioning

Systems such as IBM's 370/VM and VMware's GSX and ESX servers are, in a sense, techniques to partition the system and provide multiple operating system images on a single hardware platform. We discussed such systems extensively in **Chapter 8**. From the point of view of isolating one operating system from another, these systems are similar to logical partitioning—software failures can mostly be confined to a guest single operating system image; they do not bring down other guest operating system images. However the VMM in these systems is a larger body of software, and the chances of an error in the VMM are higher than the chances of an error in the hypervisor of a logically partitioned system. In order for a system VM to provide cluster virtualization capability, it is desirable for it also to emulate the physical fault containment aspects of a cluster or of physical partitioning. This imposes special requirements on the VMM. In particular, each group of physical nodes that emulates a multiprocessing node in the virtual cluster should run its own VMM so that a hardware failure in a node will not cause another virtual node in the cluster to fail.



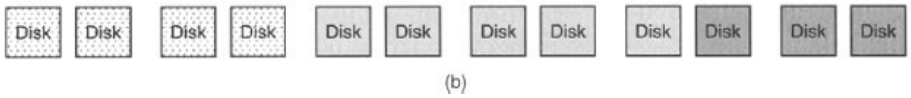


Figure 9.11 Illustration of the Effects of Dynamic Partitioning. *The figure shows the use of resources by three partitions at two different points in time. In (a) the usage of resources, for example, just after the system is initialized, it displays some degree of physical locality. However, as time progresses (b) and as the processes get migrated to achieve better load balancing, the physical range of the partitions running these processes start merging with each other.*

We illustrate the principles underlying the support of fault-containment features in a traditional system VM by describing the Cellular Disco system (**Govil et al. 1999**) developed at Stanford University, which was derived from an earlier traditional system VM called Disco (**Bugnion et al. 1997**).

9.4.1 Cellular Disco System Overview

As we saw in **Chapter 8**, a sufficient condition for efficient virtualization is the ability to intercept all the privileged operations performed by the virtual machine. Cellular Disco, which is designed to run on a MIPS multiprocessor system, achieves this by taking advantage of the fact that the MIPS system provides three levels of privilege—a user mode, a semiprivileged supervisor mode, and a privileged mode—and that most operating systems that run on MIPS systems use only two of these modes, the user and privileged modes. Cellular Disco forces the virtual systems to run in the user and supervisor modes and restricts the use of the privileged mode to the virtual machine monitor. Whenever an operating system on a virtual machine executes a privileged instruction, it traps as shown in **Figure 9.12**, because execution of such an instruction is not permitted in the supervisor mode. The trap is handled by the VMM, which then interprets the instruction. The MIPS architecture further forces all supervisor and user-mode memory accesses to go through dynamic address translation. All references to real memory made by an operating system therefore can be mapped to a region of physical memory distinct from the regions addressed by operating systems on other virtual machines.

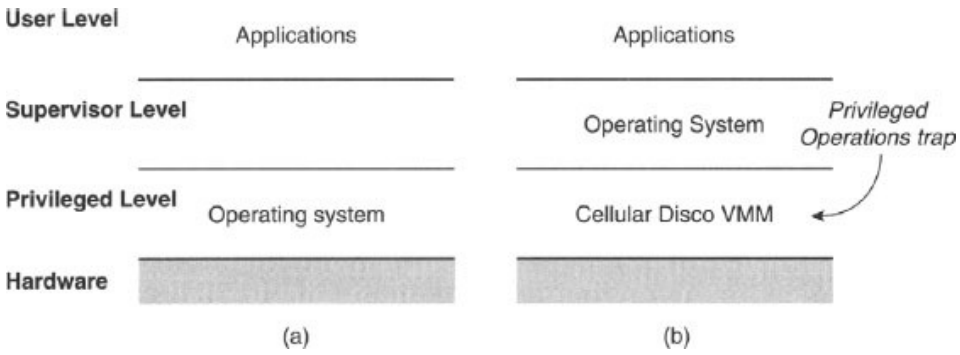


Figure 9.12 Modes in Which Various Parts of the Cellular Disco System Operate. (*a) Native system; (b) Cellular Disco.*

The virtual machine monitor also intercepts I/O requests from a guest operating system executing on a virtual machine. After checking for the validity of the request, the VMM either forwards the request to a real I/O device or handles the request itself. An example of the situation where it handles the request itself is virtual paging. In typical virtual machine implementations there are two levels of paging, one by the guest operating system and another by the VMM. It is possible for the operating system to request that a page be written out to its paging disk when the VMM itself has written this page out to its own paging disk. In a simple VMM implementation, such a request would be satisfied by first bringing back the page from the paging disk of the VMM and then writing the page out to the paging device of the guest operating system. Cellular Disco makes this process more efficient by trapping every read and write to the guest operating system's paging disk. It keeps internal data structures that map the status of all pages. Thus when the request to write out a page to the kernel's paging disk is observed, Cellular Disco notices that the page has already been paged out to its own paging disk and simply annotates the internal mapping to indicate the actual location of the page. When the guest operating system subsequently wishes to read from its paging disk, the mapping is consulted to determine the true location from where its contents are to be read.

9.4.2 Memory Mapping

The mapping of real memory in a virtual machine to physical memory is performed in Cellular Disco with the help of two maps. The *pmap* data structure is indexed by the real address and returns the corresponding physical address in the system. The *memmap* data structure maps a physical address back to the real address.[1]

Unlike the physical partitioning schemes and some of the logical partitioning schemes where the physical memory is strictly parceled out to the different virtual machines on the system and where every real memory location on a virtual machine is associated with a physical memory location, Cellular Disco operates like a traditional system virtual machine and virtualizes the real memory also. This allows the overcommitment of real memory—the sum of the real memory assumed by all active virtual machines on the system can be more than the available physical memory on the system. Each virtual machine is allocated a resident set size that is dynamically trimmed as the system begins to run low on physical memory. The VMM incorporates a page-replacement policy of its own that takes advantage of information about the usage of pages. If a page contains unneeded data, annotations in the internal data structure allow the VMM to detect this and avoid the overhead of writing the page out to disk. This would be useful, for example, when the page that needs to be swapped out is an unallocated page on a virtual machine.

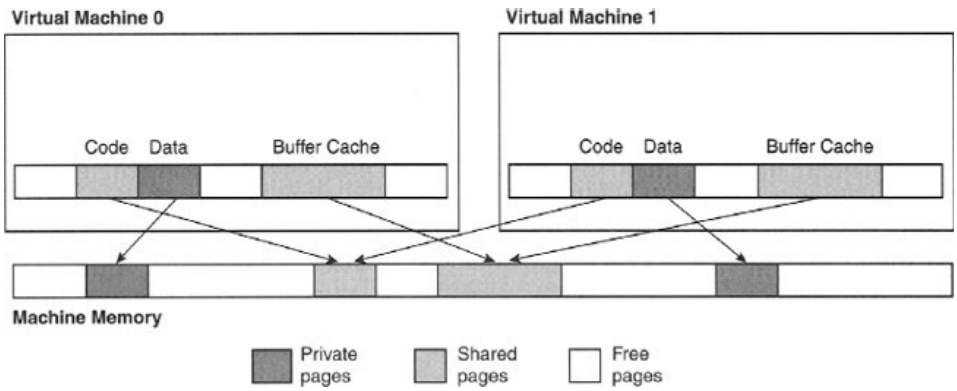


Figure 9.13 Memory Sharing in Cellular Disco.

Cellular Disco provides functions to an application beyond those normally available to the application through the operating system interface. An example is the ability to split large applications into multiple processes working in separate virtual machines that nonetheless share global regions of memory. System calls are provided to allow processes within an application to register such shared regions. The process of registration can bypass the guest operating system because all system calls have to be intercepted by the VMM. **Figure 9.13** shows two virtual machines sharing code pages as well as a buffer cache while still keeping their application data regions separate. No additional overhead is incurred in writing to a shared location because the underlying platform is a shared-memory multiprocessor, and cache coherence guarantees that other processors will see the write in the shared region in a timely manner.

This system is a convenient and efficient alternative to the conversion of a large shared-memory application to a cluster application that communicates using messaging and network protocols. The only overhead for the application is the need to relink the application with a different shared-memory library. Cellular Disco handles the paging of the shared regions to its paging disk, attaching information about all virtual machines sharing the page along with the contents of any page being written out to disk. For efficiency, it places this information in a sector contiguous to the paged-out data, thus avoiding the penalty of an additional seek of a disk sector.

9.4.3 Fault Containment

One important aspect that differentiates Cellular Disco from conventional system virtual machines is its support for hardware fault containment. Hardware fault containment is an important characteristic of clustered systems—the effects of a hardware fault in one part of the system are usually contained within the node of the cluster containing the fault. Operating systems and applications running

on other nodes usually can continue to operate unaffected. Unfortunately there are a few critical faults that could potentially bring down the entire system. For example the VMM itself, which manages all the resources of the system, is vulnerable to failure of the hardware components it uses. It is important therefore to keep such critical parts to a minimum in order to increase the mean time between failures (MTBF) of the system as a whole.

Cellular Disco treats the hardware as a set of cells, with each cell running its own VMM and managing the physical memory it contains, as shown in **Figure 9.14**. A failure in a hardware component within a cell does not affect virtual machines on the system that do not use that cell. Cellular Disco increases the MTBF of the system by keeping the monitor code compact (less than 50K lines of code) and by restricting access of the VMM in one cell to only those physical memory locations belonging to that cell. Communication with other cells is achieved through the use of a carefully designed and fast remote procedure call, which serves as an efficient and trusted interprocessor communication primitive.

One way for such a system to manage fault containment is to limit each virtual machine to operate on a fixed set of cells. The main problem with this is that it can lead to underutilization of the system because of the inability of the VMM to move the workload toward idle processors in other cells. Thus, there is a tension between the requirements of load balancing and hardware fault containment. Cellular Disco attempts to get around this problem by associating each processor with a fixed list of virtual CPUs (VCPUs) and by allowing migration, over time, of a virtual CPU from one processor to another, even across cell boundaries. The establishment of a VCPU list for each processor simplifies the scheduling of processors and eliminates a source of contention and lock complexity in conventional systems. The ability to migrate VCPUs, although at coarser time intervals than conventional systems, permits better load bal-

ancing on the system. For fault containment, rather than force the boundaries of a virtual machine to the minimum number of cells that could ideally contain it, Cellular Disco allows local deviations from such a strict policy.

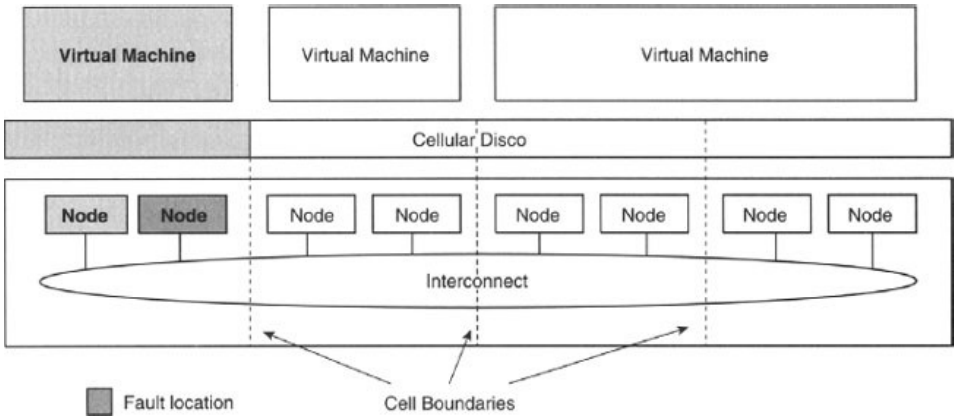


Figure 9.14 Fault Containment in Cellular Disco. *The impact of a hardware fault is contained in the virtual machines operating on the cells where the fault occurs. Shading indicates affected area due to a fault in the indicated node.*

Consider a virtual machine that needs all the CPUs contained in two cells and that has been running well for some time but suddenly finds itself slowing down because of overload on one if its assigned processors, say, due to too many active VCPUs. Cellular Disco will discover such a situation and attempt first to move the particular VCPU of the virtual machine to another, less loaded processor within the same cell. This satisfies the fault-containment requirements of the system. If a lightly loaded processor is not available within the cell but is available in some other cell, Cellular Disco allows the migration of the VCPU to another cell. Thus the virtual machine, which needed the resources of only two cells, becomes vulnerable to faults in three cells rather than the minimum two. Some degree of protection to faults is traded off for better performance on the system as a whole. More components of the virtual machine can migrate to the new cell if the new cell is comparatively lightly loaded. Eventually, if all the VCPUs migrate from one cell to the new cell, Cellular Disco provides a mechanism to eliminate completely all data and control information residing in the old cell, thus reducing the fault vulnerability to two cells rather than three

ing the fault vulnerability to two cells rather than three.

9.4.4 Memory Borrowing

A second situation where Cellular Disco allows flexibility from a rigid fault-containment policy is in the use of memory. The virtual machine monitor running on each cell manages the memory attached to that cell. This helps in fault containment but can lead to imbalance in the use of memory resources. Thus a cell that is a part of a memory-intensive virtual machine may run out of physical memory and may page in and out of disk frequently, even though other cells in the system may have plenty of free physical memory available. Cellular Disco permits relaxation from the rigid fault-containment policy by allowing a cell to temporarily *borrow* memory from other cells.

Memory borrowing in Cellular Disco is accomplished as follows. Each cell maintains a list of free pages. Initially this list contains only pages that physically belong to the cell. As the pages get used up and the cell begins to run low on physical memory, it considers borrowing memory pages from other cells. The potential candidates from which a cell borrows memory are determined by the following.

- **The list of cells that each virtual machine has designated for this purpose.** Users of a virtual machine can determine the extent to which they wish to protect themselves from faults in the system. The smaller the set the virtual machine designates, the lower is its vulnerability to faults but the higher is the probability of performance loss due to paging.
- **Cells that have already been actively supplying memory pages to virtual machines in this cell.** These cells are kept in a vulnerability list. By ensuring a minimum availability of free pages from cells in the vulnerability list, the policy biases borrowing to favor those cells that have already been supplying pages, and hence prevents an increase in the vulnerability to faults.
- **The availability of pages in the remote cells.** Cells are candidates for borrowing if they have a certain minimum amount of memory available. When the available memory drops below this amount, the cell may refuse a request to borrow pages

of its memory.

- **The overall memory requirements of the requesting virtual machine.** In order to keep virtual machines that do not need much memory from having to span multiple cells, Cellular Disco biases its allocation of available local memory in a cell to such virtual machines. Thus a request for memory when the available local memory has fallen below a threshold may actually end up being satisfied locally.

A flowchart indicating the actions taken on a page fault in a cell is shown in **Figure 9.15**. It is interesting to note that Cellular Disco resorts to paging only when memory borrowing fails. More than one page is borrowed at a time in order to reduce the number of such requests. Once a page is borrowed from another cell, the monitor puts it in its free list, tagged with the identity of the cell that it belongs to. There is no penalty to accessing this memory for routine reads and writes—the only effect of borrowing is the increased vulnerability to faults in the virtual machine that uses a borrowed page.

9.4.5 Recovering from a Fault

The flexibility provided to virtual machines for going beyond hard cell boundaries both for CPUs and memory makes the job of recovering from hardware faults more complex. Cellular Disco must ensure that all affected cells and virtual machines are taken care of in the event of a fault. The steps taken by Cellular Disco on a fault can be summarized as follows.

1. The hardware determines the extent of the fault and attempts to recover from it.
2. Through an interrupt the hardware informs all processors in the system of the recovery. This interrupt is handled by Cellular Disco, which then initiates its recovery process.
3. Cells communicate with each other to agree on a set of hardware nodes that are still functioning. The ability to share memory between cells is exploited to expedite this process. The set of nodes determined to be “live” is used to restore the communication mechanism to a state where it is “unjammed” of all messages to and from nodes that are outside the live set.
4. Each VMM on a functioning cell determines all the virtual machines affected by