

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.

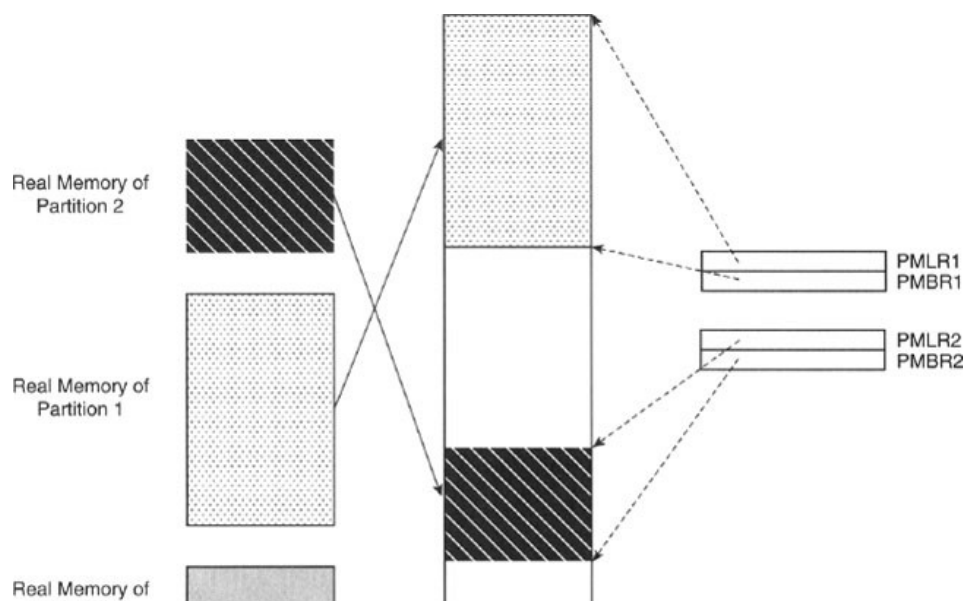
One way to do this is to partition the physical memory into disjoint regions. One of these regions is allocated to the hypervisor itself, while each of the other regions is allocated to each of the logical partitions currently active on the system, as shown in [Figure 9.7](#). Corresponding to each partition 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 to obtain the physical address. The PMLR is used to

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.



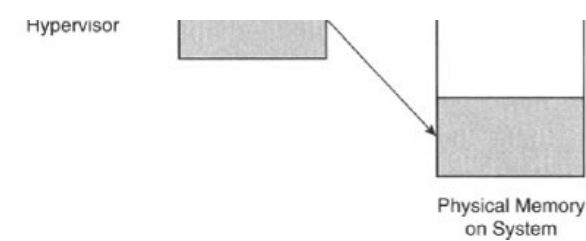
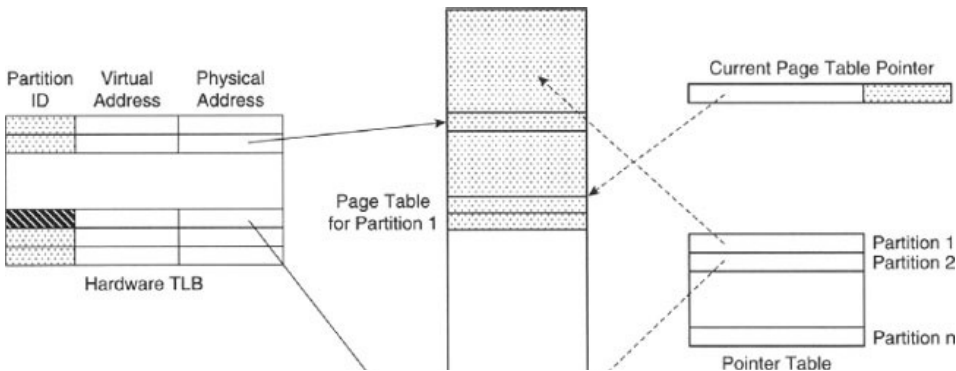


Figure 9.7 Mapping of Real Memory of Partitions to Physically Available Memory. When contiguous chunks of physical memory are allocated to each partition, the mapping table is simple and needs just two pointers as shown.

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.



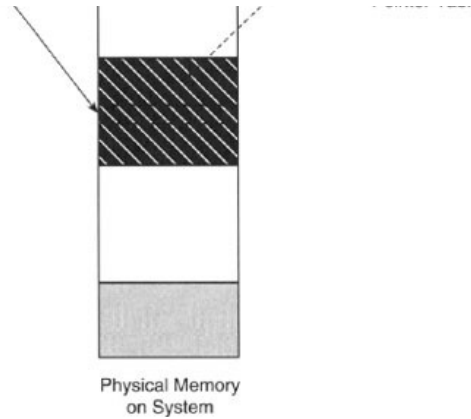


Figure 9.8 Adaptation of a Hardware TLB for Logical Partitioning. *The pointer table is typically small and can be maintained in hardware. The current page table pointer is loaded with the entry in the pointer table corresponding to the currently active partition.*

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
<i>find_valid()</i>	Find any valid entry for a given virtual address

	given virtual address
<i>Invalidate()</i>	Invalidate a specified entry
<i>add_validate()</i>	Add a new valid entry
<i>modify_pp()</i>	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.



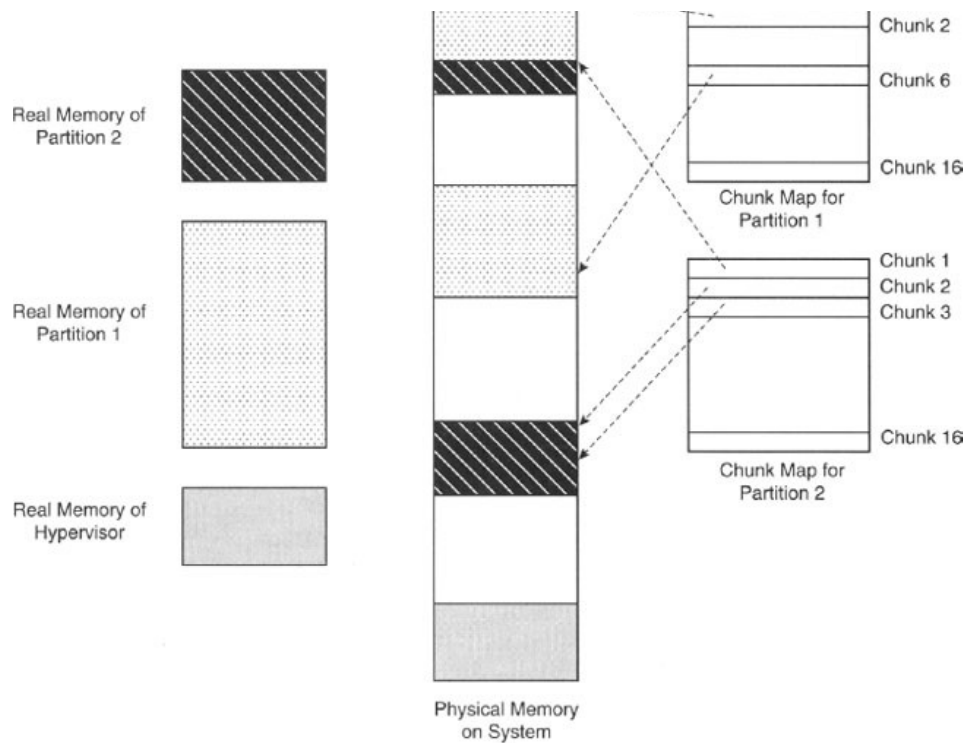


Figure 9.10 Illustration of the Dynamic Allocation of Real Address Space for Partitions Using Chunks.

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.

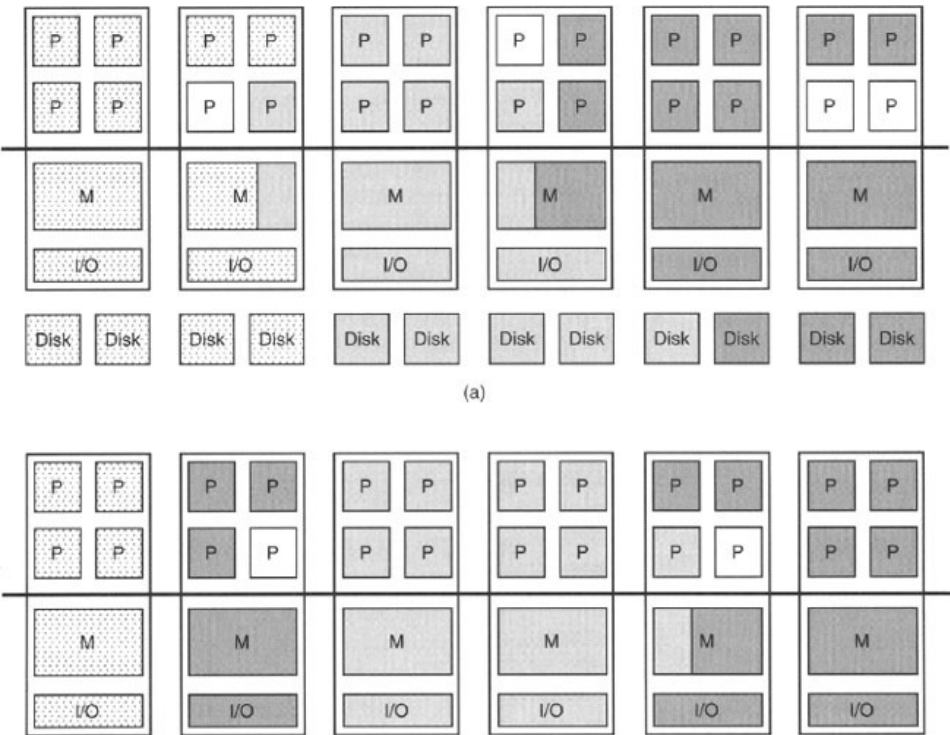
- Environmental factors, such as the power usage, temperature, or reliability of the hardware
- Business factors, such as the security of the system and the fulfillment of various system-level objectives contracted with users of the system
- Operational factors, such as efficiency of utilization of hardware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



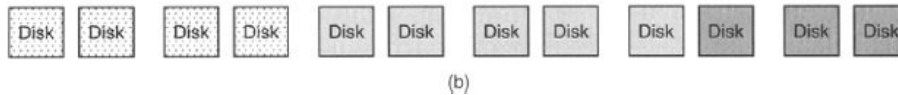


Figure 9.11 Illustration of the Effects of Dynamic Partitioning. *The figure shows the use of resources by three partitions at two different points in time. In (a) the usage of resources, for example, just after the system is initialized, it displays some degree of physical locality. However, as time 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.

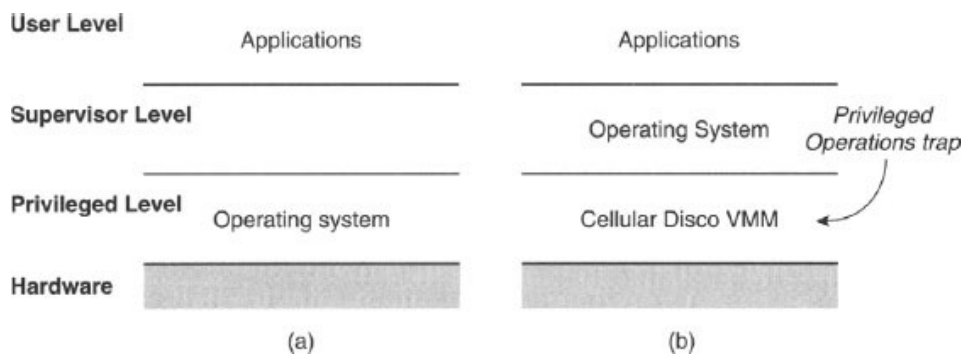


Figure 9.12 Modes in Which Various Parts of the Cellular Disco System Operate. (a) Native system; (b) Cellular Disco.

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.

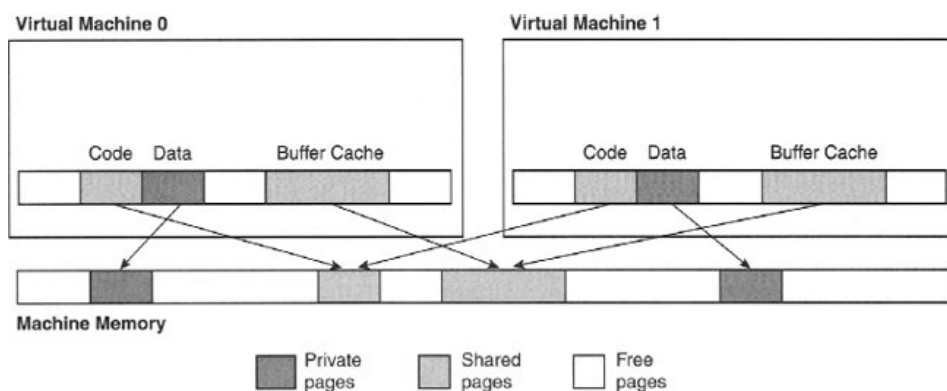


Figure 9.13 Memory Sharing in Cellular Disco.

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 inter-processor 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.

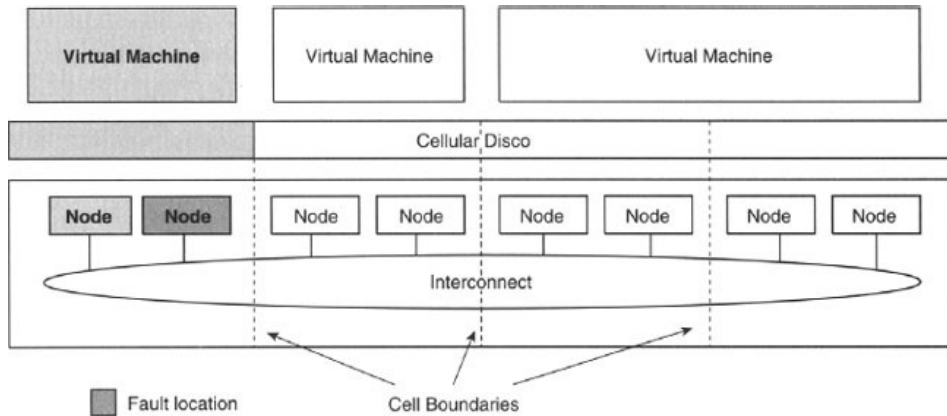


Figure 9.14 Fault Containment in Cellular Disco. The impact of a hardware fault is contained in the virtual machines operating on the cells where the fault occurs. Shading indicates affected area due to a fault in the indicated node.

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 of 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

the fault. In doing so, it consults the various internal data structures that contain information about the cells and physical resources that each virtual machine was using.

The entire system needs to participate in order to ensure that all affected virtual machines are properly purged from the system and that all references to the affected virtual machines are eliminated from the data structures kept in each cell. This is a considerably more elaborate process than that needed on a physically partitioned system—and is the cost for improving the utilization of the system under normal operating conditions.

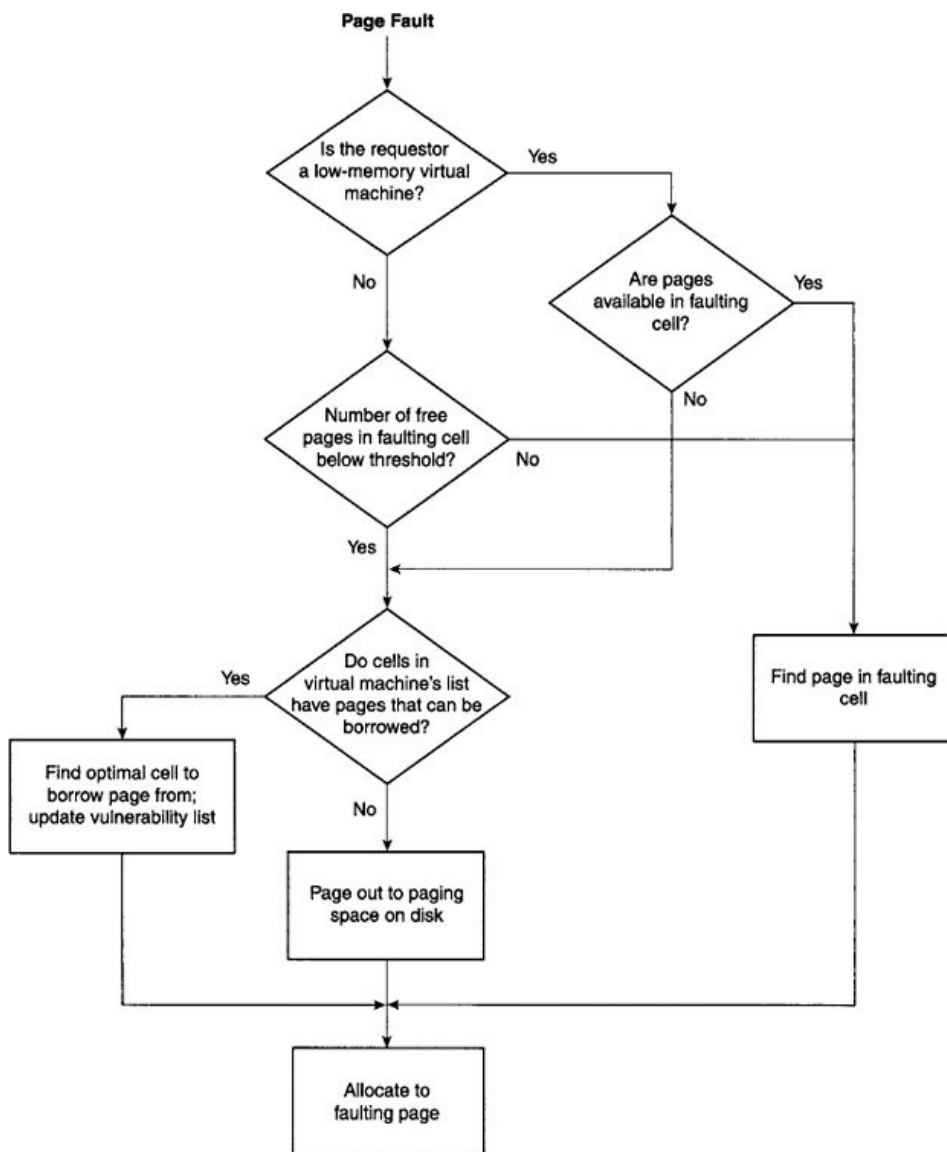


Figure 9.15 Cellular Disco's Actions on a Page Fault. *As far as possible, the system avoids using the paging disk. The requests of virtual machines with low memory requirements are*

satisfied locally, if possible. When choosing a cell to borrow from, a cell from the vulnerability list is chosen, if available.

The Cellular Disco research suggests that it is possible to implement fairly-sophisticated virtualization and fault-containment mechanisms in an efficient way. Its general philosophy of using physical boundaries as strong hints for the purposes of fault containment rather than as rigid walls provides a degree of flexibility, especially in situations where an application is willing to trade off some of the fault-containment characteristics for better performance. While the research was conducted in the context of system virtual machines, many of the ideas could eventually find application in hypervisor-based logically partitioned systems.

9.5 Virtualization with Different Host and Guest ISAs

All the schemes described thus far have involved situations where the ISA of the virtual system is identical to the ISA of the host system. When the two ISAs are different from each other, added complexities are introduced.

- The instructions of the target ISA must be dynamically emulated by the host system. The translation of instructions from the ISA of the virtual machine to the native ISA is an operation that must not be visible to the virtual machine or the programs running on it, including the operating system. The emulation techniques needed to achieve this are those described in connection with the implementation of process VMs ([Chapters 2–4](#)) for most ISA features and codesigned VMs ([Chapter 7](#)) with respect to some system ISA features, such as page faults.
- The memory model of the target system, particularly the coherence and memory ordering rules, must be observed on the virtual system. These involve additional actions on the part of the host system. The host could handle this either in hardware, in the firmware, or in emulation software contained in the trusted monitor layer, i.e., either in the virtual machine monitor or in the hypervisor. As described earlier, there are two different ways in which multi-

processors are generally configured. The processors can be connected in a loosely coupled cluster configuration, where not much other than I/O is shared between the processors. Alternatively, the processors can be configured in a shared-memory configuration, with all processors in the multiprocessor configuration sharing the same memory. Implementing a cluster virtual machine on a native cluster system is fairly straightforward if each target virtual processor is mapped to one real processor. This case is shown in **Figure 9.16**. Since memory is not shared between nodes in such a cluster, it is sufficient to virtualize each processor using techniques given earlier in this book. Virtual node communication is through messages that can be translated to similar messages on the host.

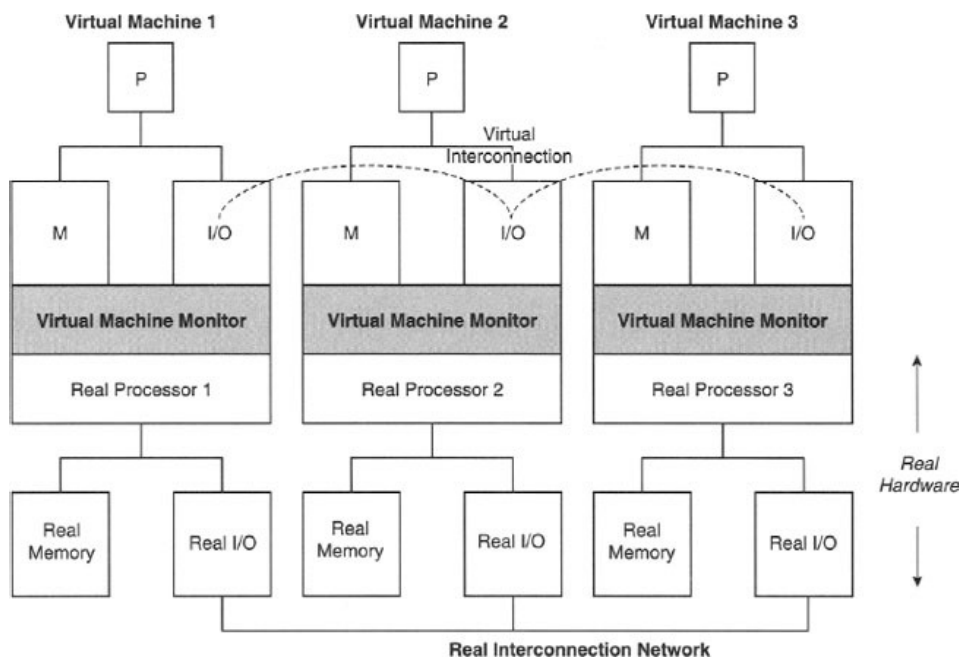


Figure 9.16 Virtual Uniprocessor Cluster on a Real Uniprocessor Cluster.

When the system to be virtualized is a shared-memory multiprocessor, the situation becomes more complex. Because the memory is shared between the processors, the monitor on each processor must handle shared-memory requests exactly the way they would be handled on a native machine. So, for example, if the guest virtual machine supports coherent shared memory, the monitors running on the separate processors must ensure that information written to any

the separate processors must ensure that information written to any shared-memory location is made visible to all processors that share that location. If the underlying native system is itself a shared-memory system, then the hardware mechanisms that already exist on the system to support coherence can be used to ensure coherence for the virtual system. This is the case, for example, in the FLEX-ES system from Fundamental Software Inc. (FLEX), which provides a virtual SMP for an IBM System/390 system as a user application under Linux on an Intel IA-32 based SMP. The basic IA-32 SMP is enhanced with special hardware adapters, as shown in [Figure 9.17](#), to emulate the functions of the System/390 communication and channel I/O adapters. Since the underlying Intel IA-32 platform also implements memory coherence, the coherence expected by the virtualized mainframe system is trivially supported. While the memory ordering rules of the Intel IA-32 are not exactly identical to those of the System/390, the differences can be taken care of in the VMM layer without significant degradation of performance.

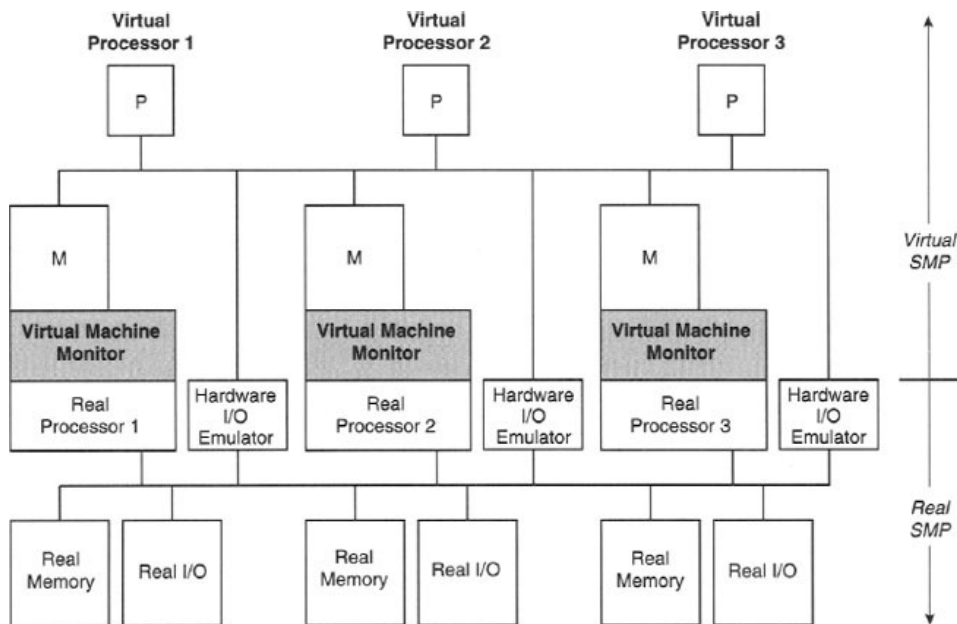


Figure 9.17 Virtual Shared-Memory System on a Real Shared-Memory System. When there is a mismatch between the I/O of the virtual system and the real system, it may be necessary to add hardware, as shown, for efficient I/O emulation.

If the host's model does not match the guest's memory model, then it becomes an additional task of the combination of emulation soft-

becomes an additional task of the combination of emulation software running on all processors to communicate with each other and keep track of shared locations. Most solutions to this problem require several instructions and several cycles for each coherence request. Hence, when there is a modest amount of sharing in a system, the emulation performance begins to degrade considerably. The next section examines some ways to handle such cases and discusses the potential degradation of performance. Only a cursory treatment is provided here to this rather complex problem.

9.5.1 Memory Model Emulation

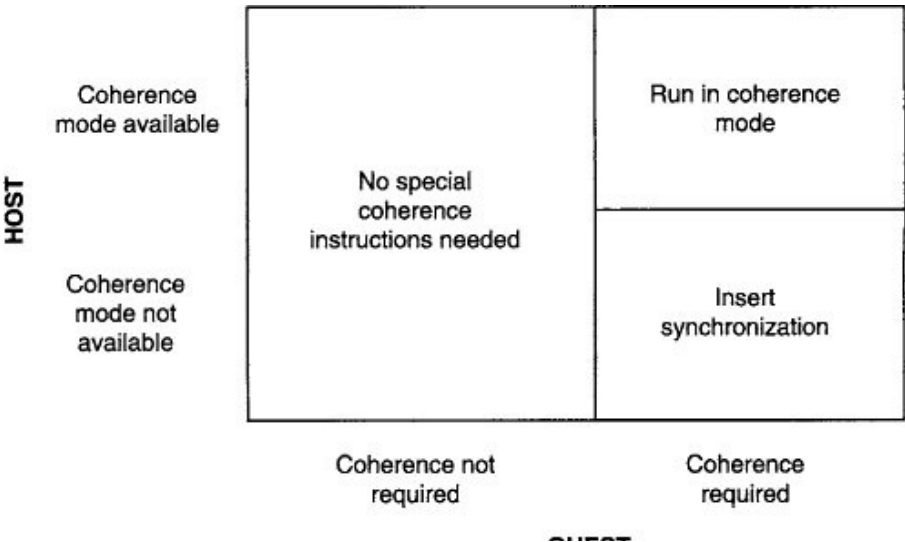
As described in Appendix A, the ISA of most processors includes special conventions for the access of shared-memory locations. As with all aspects of an ISA, these conventions represent a contract between the programmer and an implementation of the ISA. A programmer writes a program assuming that accesses to variables shared among processors obey certain rules, while the hardware designer (the processor designer or the system designer) ensures that the rules used by the programmer are indeed guaranteed by the implementation. We note two main aspects of the memory model, the memory coherence and the memory consistency models, that determine how results appear in different shared-memory systems. These are the two aspects that need to be given special consideration when a multiprocessor system is virtualized—the differences in these models between the host and the guest ISAs can produce undesirable results during virtualization, unless special actions are taken by the emulation software.

Memory Coherence Emulation

Recall that memory coherence is said to be implemented on a multiprocessor system if the order of writes to a given location by one processor is maintained when observed by any other processor in the system. Appendix Section A.7.3 describes the difference in be-

havior of coherent and noncoherent systems. The nature of actions that emulation software must take to support the coherence model of a guest virtual machine depends on the coherence models of both the guest and the host. We list the various possibilities in **Figure 9.18**.

- 1. Memory coherence is not required on either the guest or the host:** In this case no action is required because programs have presumably been written in a way that accounts for this absence of coherence. Synchronization in such systems is done through explicit synchronization instructions. The synchronization primitives in the host and the guest may not match, but it is usually not difficult to emulate a synchronization primitive of the guest using synchronization primitives of the host.
- 2. Memory coherence is required by the guest and is available on the host:** In this case, the emulation software ensures that the virtual machine runs with the memory coherence option enabled.
- 3. Memory coherence is not required on the guest but is enabled on the host:** An example of this is a distributed cluster system that needs to be emulated on a shared-memory multiprocessing system. There is typically no special action that needs to be performed by emulation software. The coherence hardware, which provides a coherent view of memory across the processors, does more than is needed by the coherence model of the guest. If there is a performance disadvantage because of the coherence requirement and there is a faster noncoherent mode available on the machine, this mode can be used, as in case 1 earlier. Synchronization instructions will need to be emulated using the synchronization primitives of the host, even if the host runs in coherent mode.



GUEST

Figure 9.18 The Various Actions Needed for Emulating the Coherence Model. *In all cases synchronization primitives of the guest must be emulated on the host.*

4. Memory coherence is required on the guest but is not supported by the host:

An example of this is a shared-memory guest machine running on a distributed-memory host machine. The only option here is to use software to enforce coherence. As we said earlier, even when machines do not provide coherence support, they do provide synchronization primitives so that software can perform explicit synchronization. Unfortunately, the brute-force technique of synchronizing after each memory write by a processor is too slow, because synchronization operations, especially on large multi-processor systems, are slow. A more practical approach is to use a directory that registers the set of processors sharing various locations, with a coherence operation being triggered on the set of sharing processors when one of the processors in the set changes the contents of the shared location. This operation can get expensive as the size of the directory gets large. The size of the directory can be reduced by increasing the granularity of an entry in the directory to be a page. Keeping such information at page granularity is also convenient because the available page table structures can be used for storing sharing information and to trigger a coherence action. There have been several techniques of this nature that have been developed in connection with distributed shared-memory (DSM) computing (**Hennessy, Heinrich, and Gupta 1999**), which also deals with the problem of providing coherence to an application while running on noncoherent hardware.

Fortunately most ISAs today provide coherence support. Hence, coherence emulation is not a source of performance degradation for such virtual machines.

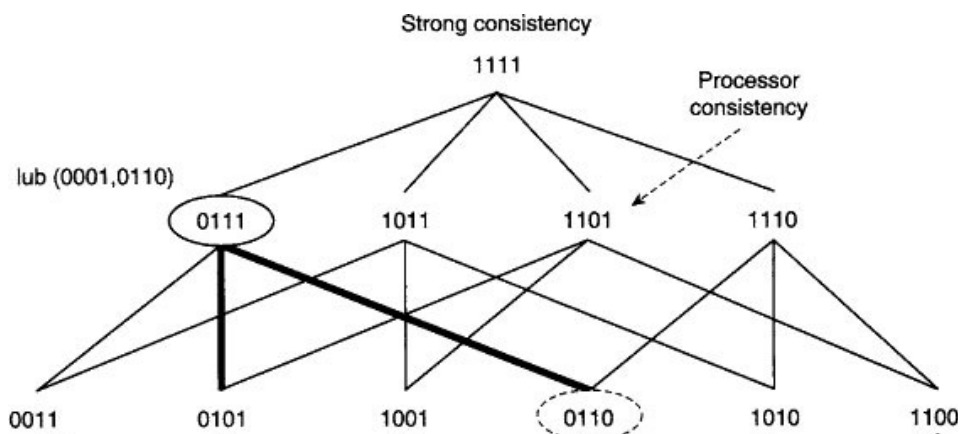
Memory Consistency Models

Memory consistency deals with the order in which accesses by one processor to different locations in memory are observed by another processor (refer to Appendix Section A.7.3 for a more detailed explanation). This is in contrast to memory coherence, which deals with the order of writes to a single location. Memory consistency deals not only with different locations but with all accesses, whether reads or writes. The important thing to take care of when a program is being run on a virtual machine is that the ordering of reads and

writes must be such that it *could have been* produced on a native implementation of the virtual machine. Thus the emulation software should account for differences between the memory consistency models of the host and guest and ensure that the ordering of memory operations is consistent with what the ISA of the guest specifies.

As noted in Appendix Section A.7.3, four hazards have to be considered in a consistency model: the read-read (RR) hazard, the read-write (RW) hazard, the write-read (WR) hazard, and the write-write (WW) hazard. In a strong consistency model, an implementation must ensure that none of these hazards is present. When this requirement is relaxed, we get one of the many relaxed consistency models. Such relaxed models are referred to as being weaker than the strong consistency model.

We could encode the consistency model using four bits, each bit indicating whether ordering is imposed for the corresponding hazard pair. So, for example, a (1111) will indicate strong ordering, while a (1101) will indicate that the WR ordering is relaxed from strong ordering, as in the processor consistency model used on the Intel IA-32 or the IBM System/390. The weakest consistency model (0000) imposes no ordering between any pair of read-write operations. Note that reads and writes to the same location by a processor must be ordered according to program order in all cases (refer to Appendix Section A.7.3).



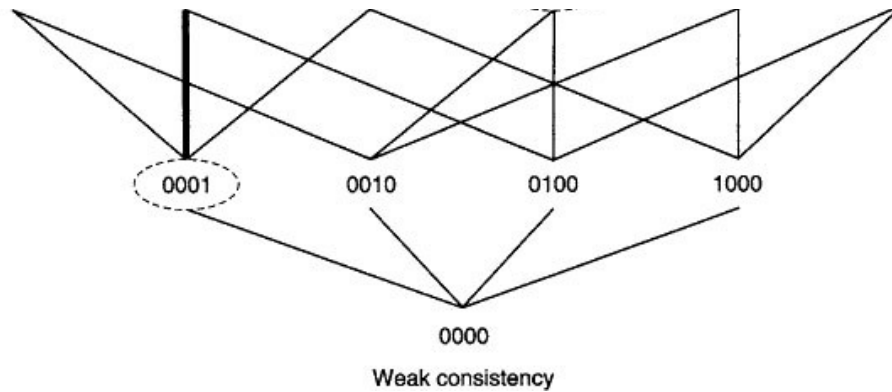


Figure 9.19 Lattice of Consistency Models. Each 4-tuple represents the four hazards *RR*, *RW*, *WR*, and *WW*. A “1” in a position indicates that the corresponding hazard is not tolerated, while a “0” indicates that the hazard is ignored by the consistency model. The 4-tuple (0000) represents weak consistency, (1111) represents strong consistency, and (1101) represents processor consistency.

For purposes of comparing two consistency models, it is convenient to draw a lattice, as shown in [Figure 9.19](#), where each node in the lattice corresponds to a consistency model. The least element on the lattice (0000) represents the weakest consistency model, while the greatest element (1111) corresponds to the strongest consistency model. When comparing two consistency models, we determine whether the least upper bound (*lub*) of the pair is itself one of the pair. If it is, then this *lub* of the pair is defined as being *stronger* than the other. Thus between (1101) and (1001), the *lub* is (1101), implying that (1101) is stronger than (1001). The model represented by (1111) is stronger than any other consistency model, because it is the greatest element in the set.

As seen from [Figure 9.19](#), the nodes (0001) and (0110) are not comparable. One model is stronger in some respects and the other stronger in other respects. There are two models that are stronger than both these models, (0111) and (1111). The *lub* of the pair is (0111). Any ordering that satisfies the *lub* (0111) therefore will also be consistent with the ordering specified by the models in the pair (0001) and (0110).

When emulating a consistency model of a guest using a different consistency model for the host, we can divide the space of possibili-

ties into the following three categories.

1. **The memory consistency model of the guest is the same as that of the host or weaker than that of the host:** In this case there is nothing special that needs to be done by the emulation software as long as the interpretation or translation process does not eliminate or reorder memory accesses to shared locations by a processor. This last restriction ensures that the order of accesses to memory in the virtual machine system belongs to the set of orderings of accesses to memory allowed by the consistency model of the virtual machine. Note that this restriction prevents the elimination of memory accesses—an access to a memory location cannot be safely eliminated just because its contents are available in a general-purpose register, for instance.
2. **The memory consistency model of the guest is stronger than (more restrictive than) the memory consistency model of the host:** In this case, it is important that the additional hazards that must be avoided on the guest must also be avoided on the host. We discuss this case in more detail in the next section.
3. **The memory consistency model of the guest is stronger in some respects and weaker in other respects:** In this case we first find the lub between the two consistency models. The lub consistency model must be stronger than the consistency model of the guest. Hence we would be emulating the guest correctly, though conservatively, if we assumed that the consistency model of the guest is the lub model. Since the lub model is also always stronger than the host model, the problem is now reduced to an example of case 2. So, as a fictitious example, if the memory model of the guest is (0110), which relaxes the RR and WR hazards, and the memory model of the host permits all hazards except the WW hazard (0001), then the lub is (0111). Thus in this example, the ordering satisfied by model (0111) would be consistent with the ordering allowed on the guest. Hence it is sufficient to accurately emulate this lub model.

Emulating a Stronger Consistency Guest

When a guest virtual machine has a consistency model that is stronger than that of the host, there are certain hazards that may occur in the host that would violate the memory consistency model of the guest ISA. Let us, for example, consider the case of a guest with a processor consistency model and a host with a release consistency model. This is an interesting case because there are several processors that belong to each of these models—a multiprocessor virtual machine implementation of IA-32 running on a PowerPC multiprocessor would be an example in this category. In this case, it is impor-

cessor would be an example in this category. In this case, it is important that any RR, RW, and WW hazards not permitted in the processor consistency model also not be permitted on the virtual machine.

Most ISAs that implement relaxed forms of consistency also provide special *memory barrier* instructions that permit the programmer to specify the points at which ordering needs to be maintained, hence essentially raising the level of granularity at which consistency is implemented. For purposes of illustration, let us define a *membar* instruction, which, when inserted into the code, ensures that there are no hazards between any pair of instructions across the barrier. Obviously then, a trivial solution for any consistency emulation is for the virtual machine emulator to place a *membar* instruction after every access to memory, as shown in [Figure 9.20a](#) and b. This ensures that for every pair of memory accesses on any given processor in the host, there is an intervening *membar* instruction that prevents a hazard between the pair. Notice that the same effect could have been obtained by placing a *membar* operation before each memory access.

Unfortunately, the *membar* instruction is an expensive instruction in terms of its execution latency. This can be appreciated better if one understands that the effect of the *membar* instruction is to ensure that all processors in the system have been informed and have acknowledged that some processor wishes to impose such an ordering. Each processor then takes appropriate action to ensure that the request is honored. Since loads and stores form roughly a third of all instructions in an instruction stream, the performance of the virtual machine can be degraded quite significantly because of the insertion of these barrier instructions. It is therefore important for the monitor to restrict the number of *membar* instructions whenever possible.

Notice that the trivial solution just demonstrated works for any consistency model on the guest and, in particular, for the strong model.

The first optimization that can be performed is to eliminate those memory barrier instructions where the memory model allows relaxation. In the case of processor consistency, the memory barrier instruction after a write can be eliminated as shown in [Figure 9.20c](#) if the subsequent memory access is by a read instruction. The second optimization that can be performed is to eliminate a memory barrier instruction after an instruction if the next sequential memory access is to the same location as the previous memory access—program order requirements take care of these even without barrier instructions. This must, however, be done with care. In the example of [Figure 9.20c](#) the *membar* instruction following Read A cannot be eliminated because the *membar* instruction just after Write A has been removed, thus permitting the following Read B to be done before Write A. However, processor consistency does not permit Read B to be done before Read A, implying that there must be at least one *membar* between these two instructions. Notice that the *membar* that was between the Read C—Write C can be eliminated without danger of the Write B moving ahead. It is important to note that there are other ways of inserting *membars* in the code that achieve the same objective.

Accesses in guest (processor consistency)	Accesses in host (release consistency)	Eliminating membars in host (W-R relaxation)
...	...	...
Read A	Read A	Read A
...	membar	membar
Write A	...	...
...	Write A	Write A
Read B	membar	...
...	...	Read B
Read C	Read B	membar
...	membar	...
Write C	...	Read C
...	Read C	...
Write B	membar	Write C
	...	membar
	Write C	...
	membar	Write B
	...	membar
	Write B	
	membar	

(a)

(b)

(c)

Figure 9.20 Using *membar* Instructions to Impose Consistency Requirements of the Guest That Are Stricter Than That of the Host. (a) Original host code. (b) Conservative use of *membar* instructions after each memory access. (c) Reducing the number of *membars* to account for W-R relaxation in the guest

could for write emulation in the guest.

This example shows that it is difficult to reduce the overhead of inserting memory barrier instructions when a guest has a stronger consistency model than the host. However, there is another way to reduce the overhead of such emulation. Even in a multiprocessing system there are many occasions when a processor executes a single threaded application or a portion of a multithreaded application that does not share memory with other threads. Since the hazards we are trying to avoid are multiprocessor hazards, there is no need to add barrier instructions if there is some way for the virtual machine monitor to know that a thread does not share memory with an active thread on another processor. The translator in such systems is free to move memory accesses relative to each other, as long as it provides a way to recover the state of the virtual machine processor whenever it is needed.

Accesses in guest (processor consistency)	Location of membars in host (Location B is not shared)	Location of membars in host (Locations A and B are not shared)
...	...	...
Read A	Read A	Read A
...	...	...
Write A	Write A	Write A
...	...	...
Read B	Read B	Read B
...	membar	...
Read C	Read C	Read C
...	...	...
Write C	Write C	Write C
...	membar	...
Write B	Write B	membar
	membar	
(a)	(b)	(c)

Figure 9.21 Reducing the Number of Memory Barrier Instructions Using Knowledge About Sharing Characteristics of Access Locations. *We leave the last membar in the segment, because at this point we have no knowledge of the sharing pattern of the code that follows the sample segment.*

A more interesting situation arises when an application is written in such a way that the threads comprising the application are largely independent and only occasionally share memory, perhaps as a form of communication. In these cases, it may be more efficient to restrict the insertion of memory barriers to keep the required ordering of memory access to only the shared locations, as shown in

Figure 9.21.

The figure shows that when location B is not shared, we can remove the barriers between Read A and Read B because any reordering of access to location B will not be visible to any other processor. If neither location A nor location B is shared, then the only ordering visible to other processors is the Read C–Write C ordering. No *membar* is needed between this pair because of program order requirements. Hence it is unnecessary to place a *membar* anywhere in the code.

Thus the principal obstacle to good performance is the problem of determining efficiently during translation whether an access is to a shared location or not. This is a difficult problem because, although the history of accesses by an instruction can provide some measure of confidence about whether or not that access is to a shared location, there is a lot of bookkeeping involved in determining whether locations are shared, and extra overhead involved in ensuring that a trap is generated when a previously unshared location becomes shared.

Another problem is that of *false sharing*. False sharing occurs, for example, when the emulation software detects an access to a location by two different virtual processors but does not know that the guest operating system has migrated the thread from one processor to another for load-balancing purposes. False sharing could therefore lead to the elimination of fewer memory barriers than what would be possible.

Real ISAs often allow other relaxations or impose other orderings beyond the ones mentioned here. For example, on the System/390, a distinction is made between instruction reads and data reads. The ISA does not require that instructions be fetched in the order they are eventually executed. It also allows instruction fetches to occur before the reading of operands of previous instructions. Such relax-

ations, which obviously arise from a desire to make hardware more efficient, also help in reducing the overhead of emulation.

We have seen in this section that creating a virtual machine system with an ISA different from the ISA of the native host is certainly possible, though the efficiency of such virtualization depends a lot on the mismatch between the memory models of the two ISAs. In particular, when the memory model of the host is more relaxed, due either to a weaker coherence model or a weaker consistency model, the potential performance degradation is high, unless the emulation software performs more work in localizing the coherence and synchronization mismatches. As with VM assists mentioned in [Chapter 8](#), hardware could be added to assist the emulation software in this task. An extreme example of this is for the processor to provide a mode in which it supports the stronger memory models required by the guests. There are processor ISAs, such as the PowerPC ISA, that provide both a coherence mode and a mode in which coherence may not be required. There are also processor ISAs, such as Sun's SPARC ISA, that support multiple consistency models.

9.6 Summary

Large companies and other organizations are continually grappling with the problem of reducing the number and types of computer systems they must maintain without depriving their important users of the flexibility of maintaining and operating their own system. This has led to an array of systems claimed to enable such consolidation, basically by decoupling the view of the hardware as seen by the operating system from the actual hardware that the system runs on. Virtual platforms on large servers provide users with characteristics similar to those they would have enjoyed on their own private physical systems.

This chapter examined various ways in which large multiprocessor

servers can be adapted to provide multiple virtual multiprocessor platforms to various users. We have seen how different forms of virtualization handle the task of isolating virtual machines from each other while still providing the benefits of sharing resources. Some of these solutions, such as physical partitioning, are superior at isolating virtual machines from hardware failures, while others, especially the codesigned hardware–software approaches, excel in the degree of flexibility they provide in the allocation of resources to the virtual machines.

The biggest impediment to the adoption of new instruction set architectures is the existence of large bodies of application programs in binary form for the existing ISAs. Large multiprocessor systems of today are usually homogeneous—they incorporate processors that all execute the same ISA in their native mode. For most organizations, consolidating to a large multiprocessing system with one ISA will still leave a significant number of important applications in platforms of other ISAs that need to be supported, at least until they are ported to the native ISA. Hence virtualization of these other platforms will remain an important subject, at least until a more widespread adoption of platform-independent languages, such as Java, and platform-independent operating systems, such as Linux. We have seen that efficiently emulating applications of one ISA on another, while already difficult for uniprocessor platforms, is quite daunting in the case of multiprocessor systems, especially when there are incompatibilities in the memory consistency model. It is likely that new techniques will be developed, particularly exploiting thread-level parallelism, to improve the performance of emulating one ISA on another.

Same-ISA virtualization, on the other hand, appears to be here to stay. While the advantages of virtualization listed in the introduction to this chapter are compelling for enterprise servers, it is likely that smaller servers, such as those of Internet service providers, will

also begin to employ virtualization, particularly as a means of ensuring privacy between users and as a means of protecting users from malicious or accidental failures of the platform employed by other users. Indeed, as more applications use network resources for their execution, the need to protect one application from another, even in a single-user environment, may eventually lead to the adoption of virtualization techniques on personal devices such as laptop computers or even mobile phones.

1. Note that the Cellular Disco designers refer to the real memory as the physical memory of the virtual machine and to the actually available physical memory of the system as the machine memory.

Previous chapter

< [Chapter 8: System Virtual Machines](#)

Next chapter

[Chapter 10: Emerging Applications](#) >

Chapter Ten

Emerging Applications

With ever-expanding computer applications and their associated demands on computer system administration, the role of virtual machines and virtualization are likewise expanding. Virtualization enables the introduction of new system capabilities without adding complexity to already-existing and already-complex hardware and software. These new capabilities will support innovative computing paradigms and will provide critical functions in areas that have assumed major importance in a world grown increasingly dependent on computers.

In an argument reminiscent of those given for the original development of VM systems on mainframes, it has been suggested that operating systems and applications running on real machines today should be relocated to virtual machines. Part of the motivation for this is the desire to allow new services to be developed in a layer between the hardware and the operating system without modifying or trusting either the applications or the operating system. Examples of new services are intrusion-detection systems that help in isolating users from malicious attacks, environment-migration services that help mobile users access their entire computing environment without having to carry their machines everywhere, and enhanced reliability through redundant execution of a program on multiple virtual machines. There have been several efforts that use virtual machine techniques to emulate one ISA on another. Virtual machines are being used to package entire environments and applications to avoid the time-consuming job of installing operating systems and applications on machines. They are also being used to isolate data on a system, thereby allowing the same physical machine to work with confidential and public data without compromising security.

idential and public data without compromising security.

To illustrate the importance of virtual machine technologies to future systems and paradigms, in this chapter we describe three examples of emerging virtual machine applications that can make a fundamental difference in the future of computing. The first of these is the application of virtual machines to computer security, an area receiving increasing attention. Modern computing systems are some of the most complex structures ever created. With the commoditization of both hardware and software and with widespread use of network computing, it has become easier for intruders to cause damage on a large scale by taking advantage of the nearly inevitable loopholes and flaws in such complex, constantly evolving systems. Eventually, systems will mature and become more immune to attack, and it is likely that integrated mechanisms will be developed specifically aimed at thwarting such attacks. The virtual machine intrusion-detection mechanisms described in this chapter promise to be useful tools in the development of such attack-immune systems.

The second application we consider, the migration of complete computing environments, is also motivated by the commoditization and networking of computing systems. A growing number of computer users depend on having their full environment, including all data, programs, and system personalization, readily available at multiple locations (e.g., at work, at home, and while traveling). In order to do this, however, these users must carry their environment with them on a portable machine. It is more desirable, however, both for convenience as well as for security, to allow a user to log in to any computer and immediately reproduce the user's environment on that specific computer. Virtual machine technology, which provides the ability to capture the entire state of a computer system, facilitates migration of a full computing environment. Eventually, users may become accustomed to having their data and computing environment on large, remote computer farms. While such a paradigm may

obviate the need for having an entire computer at each location to which the user travels, it will not eliminate the need for migrating entire user environments from one system to another. Load-balancing, system-utilization, and system-latency issues will require means for capturing the state of the user environment and making it portable.

The notion of computing as a service or as a utility is beginning to gain acceptance. The scientific community has been actively promoting this paradigm for some time, and it also appears to be a concept that will be useful for businesses. The concept of a computational *grid* promises a seamless way through which computing resources can be shared by multitudes of users. The grid virtualizes computing resources in a way that is philosophically similar to the virtualization of resources in system virtual machines, though technically it achieves virtualization by redefining the way applications are written. In a sense, the grid concept represents the ultimate evolution of virtual machines.

Just as the Java platform fundamentally embodies virtualization, portability, and safety in its definition, the grid fundamentally embodies these same characteristics at the system and application levels. Virtualization is not optional on the grid—applications running on the grid must accept the notion that the physical characteristics of the real resources they are using are unknown to them. The hardware and operating system of the individual machines have the responsibility for providing this implementation-independent view. Although today this is done by loading extensions on existing systems, it is likely that, as in the case of security features mentioned earlier, support for operation on a grid will be integrated with the design of machine hardware and basic system software. This integration will likely include many of the virtual machine concepts described in this book.

We begin our survey of emerging VM applications in **Section 10.1**

with a discussion of the role that virtual machines can play in maintaining security in our increasingly vulnerable computer systems. In **Section 10.2**, we examine the idea of virtual machines as the enabling technology for migrating entire environments from one computer system to another. Finally, in **Section 10.3**, we take a look at the emergence of the grid computing paradigm, its similarities to other virtualization techniques, and its potential role in making virtual machines ubiquitous.

10.1 Security

Users of commercial systems, especially mainframe computers, have long been concerned with system security. These machines, which are used by large institutions, including banks, airlines, and federal agencies, handle sensitive information and are required to be secure against attacks. Security has not been as much of a concern, however, for users of common commodity computers—until relatively recently. Today, with the increase in the number of security attacks as well as the increasing cleverness of the perpetrators, computer security has become a major issue in all types of computing, including low-end desktop machines, portable machines, and even personal digital assistants (PDAs) and cellular phones. Seldom does a month pass when we do not hear about (or, even worse, fall victim to) yet another worm or virus that has invaded many thousands (if not millions) of computers around the world.

The most common way the security of a system is compromised is simply for an attacker to get access to the privileged part of the system, e.g., as a superuser in Unix environments or as an administrator in Windows systems. Interestingly, a large number of attacks are carried out by people who have gained access to such privileges without much effort—for example, through inadequately protected passwords or, more commonly, through repeated attempts to gain access to the system via commonly used passwords. Once the at-

access to the system via commonly used passwords. Once the attacker gains access to the system, he or she can tamper with the operating system to gain confidential information or even to destroy data. Often, during the initial entry into a system, an intruder simply modifies the system to ease entry into the system in the future.

Another common type of attack exploits inherent weaknesses in system software. The size and complexity of modern system software, especially the operating system, makes it very difficult to ensure that every security hole is eliminated before the system is shipped to customers. Programs written in languages such as C, which are considered “unsafe,” often exhibit bugs (or possess security holes resulting from poor programming practices) that can be exploited to gain access and hence control the system. One common type of security hole involves unchecked accesses to C arrays. When such an array is used as a buffer for user data, the user can fill the buffer with a block of data that is bigger than the array size and cause memory areas adjacent to the buffer to be overwritten. If one of these overwritten areas happens to be an address to which the program jumps, an attacker, through clever calculation, can cause that address to point to a routine, e.g., a shell program, by which the attacker gains control of the system. This is illustrated in **Figure 10.1**. It is generally not likely for an innocent user inadvertently to do harm to the system in this manner because the probability that arbitrary user data will cause a jump to a sensitive location is low. However, such security holes can be exploited maliciously and cause havoc in systems.

10.1.1 Intrusion-Detection Systems

In practice, security holes due to bugs or poor programming, as described in the previous section, eventually get fixed. However, as new code is introduced to a system, new holes appear. Clearly the most effective measure one can take to ensure the safety of a system is to isolate the system from all potential attackers (which is often

the same as isolating it from all other systems). This is not a practical solution for most systems—users are dependent on communication with other systems either over local-area networks or over the Internet, and this makes them vulnerable to malicious attacks of the kind mentioned earlier. Using object-oriented programming with built-in type and range checking as is done in Java and MSIL will also provide a high level of security, but only for those programs running within the HLL VM framework. A significant amount of system code and a number of native libraries are likely to remain outside the secure framework for quite some time.

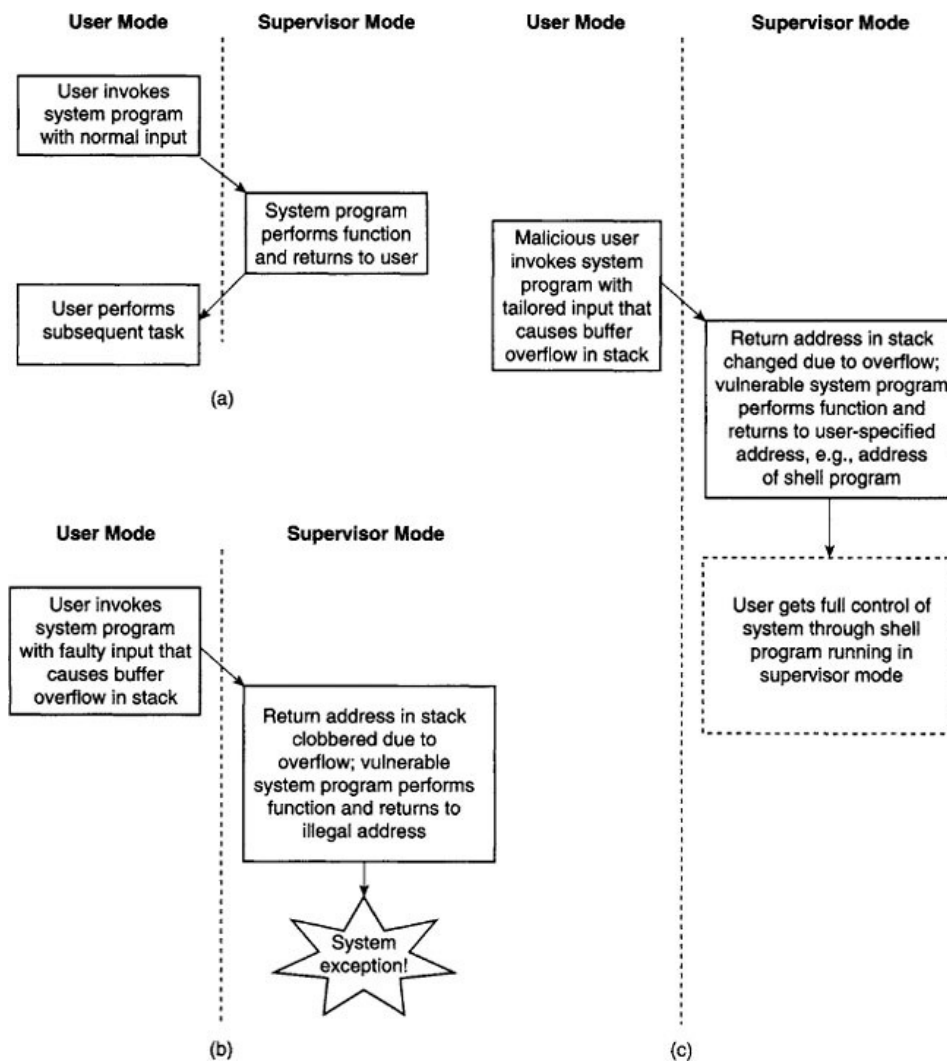


Figure 10.1 Exploiting Buffer Overflow in a Program Stack. In the normal course of events (a) the user parameter fits in the buffer. The return address on a stack is that of a location in the user's program. If the user inadvertently causes an overflow of the buffer (b), the return address may be clobbered (overwritten) but typically not with a legal instruction address. In this case, the return causes an exception. A malicious program (c) can manipulate

the parameter to be copied into the buffer so that it overwrites the return address with a location that allows the adversary to launch another valid program. This program can be a shell program that, now running in the supervisor mode, allows the user free rein over the system.

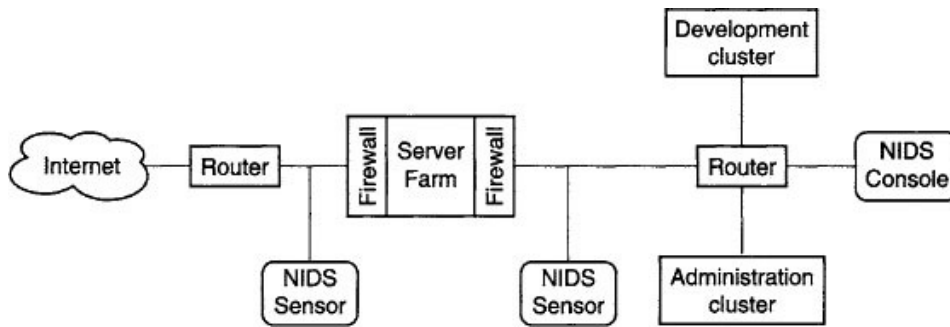


Figure 10.2 Typical Enterprise Network Intrusion-Detection System (NIDS). The sensors monitor the network for spurious activity. The console allows the programming of the sensors, logs the activity seen by the sensors, and forms the control point to isolate parts of the system when malicious activity is detected.

A common way of providing security protection for current systems is through the use of an *intrusion-detection system* (IDS). As the name suggests, these systems attempt to examine a computer either continuously or periodically to check for potential attacks. These systems depend on the general knowledge of how potential attacks may take place. Once the characteristics of an attack are known, an IDS can examine the activity on a computer to determine whether an attack with the given characteristics is occurring and then take action to shut down any malicious processes before they can cause significant harm. Since most attacks are launched from the network to which a system is connected, an intrusion system can be located at some point in the network away from the computer—in this case it is termed a *network-based intrusion-detection system*, or NIDS for short. If, on the other hand, the intrusion system is located within the computer itself, for example, in the operating system, it is referred to as a *host-based intrusion-detection system*, or HIDS.

Network-based intrusion systems (**Figure 10.2**) work by examining data packets as they move through the network, looking for specific patterns, or *signatures*, of suspicious activities. The traffic can be monitored either at the host or at points more remote from the host, e.g., at firewalls or routers. As soon as suspicious activity is detected,

a NIDS can block the activity or reroute the traffic to a quarantine location. Since it is possible for attacks to be disguised, there is no guarantee that malicious activity will always match one of the signatures in the NIDS database. There is also the possibility of innocuous traffic being deemed suspicious because of a similarity in its signature to one in the database (i.e., a *false positive*). Still, the NIDS has proven to be quite effective in many situations, particularly because it is decoupled from the system being protected.

Host-based intrusion-detection systems work by directly examining activity on the host and taking advantage of detailed knowledge of the host's operating system. A HIDS will look, for example, for repeated attempts to login by an intruder, or it will look for attempted accesses to files that should normally not be accessed by a user. Typically, the HIDS is integrated with the operating system and has high visibility over all the actions that take place on the system. Even the HIDS that operates as an application on the host enjoys high visibility by interacting closely with the operating system. In fact, such decoupling of the HIDS from the operating system can provide more protection to the system from intrusions that attempt to attack the HIDS itself.

Thus it is clear that the two types of systems are different in several respects and have their own advantages and disadvantages. A network-based intrusion system has significantly less visibility than a HIDS and has to base its decisions almost entirely on an examination of network traffic. An attacker thus has more flexibility in maneuvering around the detection system, especially by camouflaging itself. On the other hand, the NIDS can continue to monitor activity even after a host system has been successfully attacked—this is invaluable in learning about the nature of the attack and in taking measures to prevent it in the future. A system incorporating a HIDS can become completely disoriented after an attack, perhaps even providing misleading information while giving the intruder enough time to marshal additional resources to gain control of the system.

10.1.2 Monitoring and Recovering from Attacks

Attackers are constantly finding new ways to get around the defenses of a system. Thus while it is important for systems to have good intrusion-detection systems, it is equally important to take adequate measures to recover from an attack and to prevent similar attacks in the future. One important part of this effort is to perform *logging*. Logging saves information about critical activity on a system, e.g., login attempts and accesses or changes to certain important files on the system. This type of limited information can help in the analysis of methods used by an intruder and hence help in developing defenses for future attacks. However, in order to be able to recover fully from an attack, it is necessary to save much more information, for example, a checkpoint of the state of the system at some point before an attack occurs.

Many techniques and policies have been proposed for logging of events in a system. Most of these techniques assume that the operating system is functioning correctly, because they use the services of the operating system in order to record various events that occur. In fact many of them even keep the log of activities on the system being monitored. Such systems are doubly compromised. When an attack takes place, not only does the system fall under the control of the attacker, but the log of the system may also be altered in a way that is transparent to the user of the system or in a way that prevents reconstructing the attack for building up new defenses.

There are also difficulties involved with the reconstruction of a system after an attack. The first requirement is determining a known good state of the system before the occurrence of the attack. This implies that a checkpoint of the system must be available for some point in the past. From this state the actions taken by the system can

be replayed up to some later point, if complete knowledge were available about all the inputs into the system following the checkpoint. If the times at which these external inputs, such as keyboard activity or network activity were known, the system could be brought back step by step to a point just before the occurrence of the suspicious activity. Unfortunately, not all external input events result in deterministic changes to the system. Asynchronous interrupts occurring because of events outside the core, for example, from an I/O device, may not occur spontaneously or may not occur at exactly the same point when the system is replayed. Such events must be logged in sufficient detail so that they can be simulated accurately when the system is being replayed.

This completes a brief overview of the typical ways in which systems are attacked and of the approaches used by intrusion-detection systems to detect attacks. We have also outlined the requirements of system logs in order for a system to be able to recover from an attack. In the next section we examine how virtual machine technology facilitates these functions.

10.1.3 Role of Virtual Machine Technology

Most of the techniques needed to maintain security in a system have usually been implemented as separate programs on existing systems. It is reasonable to believe there would be significant advantages if the security features were integrated closely with the rest of the system. Yet it is important that techniques used to enforce security and monitor activity on a system be implemented in a way that separates their implementation from that of the system being attacked. This is where virtual machine technology comes in. We will look at three examples of the use of virtual machines in the area of system security.

Virtual Machine as a Sandbox

We have seen in [Chapters 8 and 9](#) how virtual machines allow the isolation of complete system environments from one another. Fault containment is an important property of virtual machines—a software failure on one virtual machine does not generally propagate to other virtual machines. Typical virtual machine systems are capable of isolating the failed virtual machine, shutting it down, and then restarting a new virtual machine without these actions being noticed by the other virtual machines in the system. This ability to isolate a virtual machine makes it useful as a tool ([Figure 10.3](#)) for closely examining the effects of an attack after it has occurred. As noted earlier, such postmortem analysis is useful in designing a defense against future attacks or even toward detecting such attacks before they cause harm.

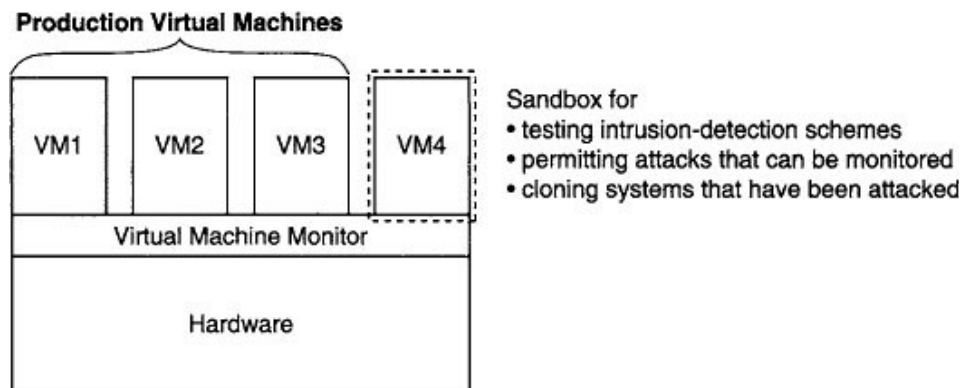


Figure 10.3 A System Virtual Machine as a Sandbox.

In a paper describing the advantages of virtual machines, Chen and Noble suggest that the complete state of a virtual machine can be saved, cloned, encrypted, moved, or restored—actions that are not so easy to do with physical machines ([P. M. Chen and Noble 2001](#)). They further suggest that the best way to understand the effects of an attack is to replay the attack and gather information about the behavior of the attack by monitoring the system as the attack progresses. While this is risky to do on a real system, one could safely carry this out on a virtual machine by cloning the original machine that was attacked, inspecting the effects of the attack, and throwing away the machine after it has served its purpose. (Cloning of virtual

machines can be done using the system encapsulation and restart techniques described in **Section 10.2**.)

Cloning a system on a virtual machine opens up other interesting possibilities. A potentially suspicious network packet or other suspicious input can be sent to a clone before it is forwarded to the actual system, to see if it has any ill effects. Similarly, during the development of intrusion-detection systems, tests for the effectiveness of the system can be conducted more safely on a cloned system running on a virtual machine than directly on the system that needs to be protected.

Virtual Machine for Monitoring Low-Level Activity

A system virtual machine forms a barrier between a system and the hardware on which it is running and also between a system and other systems running on the same hardware. This property of the virtual machine can be extended to provide a barrier between a system potentially under attack by intruders and the IDS that monitors the low-level activity of the system. Here are two possible ways to configure an IDS in a virtual machine system to achieve this.

1. The IDS can be written as a separate process in its own virtual machine, or on the host operating system of a hosted virtual machine, with special access to the functions of the VMM through a dedicated interface. This interface must support (a) means to send commands from the IDS to the VMM, for example, to enable certain monitoring functions, (b) a mechanism to efficiently access the physical memory of the system being protected, and (c) means for the VMM to transmit information about activity on the virtual machine back to the IDS.
2. The IDS can be integrated with the VMM. This gives the IDS the privilege and status of the virtual machine monitor and allows it complete access to all hardware activity of the system being protected. The VMM is a critical, central piece of software in a system and should therefore be thoroughly debugged and hardened against attacks and failures. In a scheme where the IDS is integrated with the VMM, it is imperative for the IDS also to be written, debugged, and verified with the same rigor.

Separating the IDS from the VMM makes the functioning of the IDS a