Eager et al. (1986) constructed an analytical queueing model of this algorithm. Using this model, it was established that the algorithm behaves well and is stable under a wide range of parameters, including various threshold values, transfer costs, and probe limits.

Nevertheless, it should be observed that under conditions of heavy load, all machines will constantly send probes to other machines in a futile attempt to find one that is willing to accept more work. Few processes will be off loaded, but considerable overhead may be incurred in the attempt to do so.

A Receiver-Initiated Distributed Heuristic Algorithm

A complementary algorithm to the one given above, which is initiated by an overloaded sender, is one initiated by an underloaded receiver, as shown in Fig. 8-25(b). With this algorithm, whenever a process finishes, the system checks to see if it has enough work. If not, it picks some machine at random and asks it for work. If that machine has nothing to offer, a second, and then a third machine is asked. If no work is found with N probes, the node temporarily stops asking, does any work it has queued up, and tries again when the next process finishes. If no work is available, the machine goes idle. After some fixed time interval, it begins probing again.

An advantage of this algorithm is that it does not put extra load on the system at critical times. The sender-initiated algorithm makes large numbers of probes precisely when the system can least tolerate it—when it is heavily loaded. With the receiver-initiated algorithm, when the system is heavily loaded, the chance of a machine having insufficient work is small. However, when this does happen, it will be easy to find work to take over. Of course, when there is little work to do, the receiver-initiated algorithm creates considerable probe traffic as all the unemployed machines desperately hunt for work. However, it is far better to have the overhead go up when the system is underloaded than when it is overloaded.

It is also possible to combine both of these algorithms and have machines try to get rid of work when they have too much, and try to acquire work when they do not have enough. Furthermore, machines can perhaps improve on random polling by keeping a history of past probes to determine if any machines are chronically underloaded or overloaded. One of these can be tried first, depending on whether the initiator is trying to get rid of work or acquire it.

8.3 VIRTUALIZATION

In some situations, an organization has a multicomputer but does not actually want it. A common example is where a company has an e-mail server, a Web server, an FTP server, some e-commerce servers, and others. These all run on different computers in the same equipment rack, all connected by a high-speed network, in other words, a multicomputer. In some cases, all these servers run on

separate machines because one machine cannot handle the load, but in many other cases the primary reason not to run all these services as processes on the same machine is reliability: management simply does not trust the operating system to run 24 hours a day, 365 or 366 days a year, with no failures. By putting each service on a separate computer, if one of the servers crashes, at least the other ones are not affected. While fault tolerance is achieved this way, this solution is expensive and hard to manage because so many machines are involved.

What to do? Virtual machine technology, often just called **virtualization**, which is more than 40 years old, has been proposed as a solution, as we discussed in Sec. 1.7.5. This technology allows a single computer to host multiple virtual machines, each potentially running a different operating system. The advantage of this approach is that a failure in one virtual machine does not automatically bring down any others. On a virtualized system, different servers can run on different virtual machines, thus maintaining the partial failure model that a multicomputer has, but at a much lower cost and with easier maintainability.

Of course, consolidating servers like this is like putting all of your eggs in one basket. If the server running all the virtual machines fails, the result is even more catastrophic than a single dedicated server crashing. The reason virtualization works however, is that most service outages are not due to faulty hardware, but due to bloated, unreliable, buggy software, especially operating systems. With virtual machine technology, the only software running in kernel mode is the hypervisor, which has two orders of magnitude fewer lines of code than a full operating system, and thus two orders of magnitude fewer bugs.

Running software in virtual machines has other advantages in addition to strong isolation. One of them is that having fewer physical machines saves money on hardware and electricity and takes up less office space. For a company such as Amazon, Yahoo, Microsoft, or Google, which may have hundreds of thousands of servers doing a huge variety of different tasks, reducing the physical demands on their data centers represents a huge cost savings. Typically, in large companies, individual departments or groups think of an interesting idea and then go out and buy a server to implement it. If the idea catches on and hundreds or thousands of servers are needed, the corporate data center expands. It is often hard to move the software to existing machines because each application often needs a different version of the operating system, its own libraries, configuration files, and more. With virtual machines, each application can take its own environment with it.

Another advantage of virtual machines is that checkpointing and migrating virtual machines (e.g., for load balancing across multiple servers) is much easier than migrating processes running on a normal operating system. In the latter case, a fair amount of critical state information about every process is kept in operating system tables, including information relating to open files, alarms, signal handlers, and more. When migrating a virtual machine, all that has to be moved is the memory image, since all the operating system tables move too.

Another use for virtual machines is to run legacy applications on operating systems (or operating system versions) no longer supported or which do not work on current hardware. These can run at the same time and on the same hardware as current applications. In fact, the ability to run at the same time applications that use different operating systems is a big argument in favor of virtual machines.

Yet another important use of virtual machines is software development. A programmer who wants to make sure his software works on Windows 98, Windows 2000, Windows XP, Windows Vista, several versions of Linux, FreeBSD, OpenBSD, NetBSD, and Mac OS X no longer has to get a dozen computers and install different operating systems on all of them. Instead he merely create a dozen virtual machines on a single computer and installs different operating systems on each one. Of course, the programmer could have partitioned the hard disk and installed a different operating system in each partition, but this approach is more difficult. First of all, standard PCs support only four primary disk partitions, no matter how big the disk is. Second, although a multiboot program could be installed in the boot block, it would be necessary to reboot the computer to work on a new operating system. With virtual machines, all of them can run at once, since they are really just glorified processes.

8.3.1 Requirements for Virtualization

As we saw in Chap. 1, there are two approaches to virtualization. One kind of hypervisor, dubbed a **type 1 hypervisor** (or **virtual machine monitor**) is illustrated in Fig. 1-29(a). In reality, it is the operating system, since it is the only program running in kernel mode. Its job is to support multiple copies of the actual hardware, called **virtual machines**, similar to the processes a normal operating system supports. In contrast, a **type 2 hypervisor**, shown in Fig. 1-29(b), is a completely different kind of animal. It is just a user program running on, say, Windows or Linux that "interprets" the machine's instruction set, which also creates a virtual machine. We put "interprets" in quotes because usually chunks of code are processed in a certain way and then cached and executed directly to improve performance, but in principle, full interpretation would work, albeit slowly. The operating system running on top of the hypervisor in both cases is called the **guest operating system**. In the case of a type 2 hypervisor, the operating system running on the hardware is called the **host operating system**.

It is important to realize that in both cases, the virtual machines must act just like the real hardware. In particular, it must be possible to boot them like real machines and install arbitrary operating systems on them, just as can be done on the real hardware. It is the task of the hypervisor to provide this illusion and to do it efficiently (without being a complete interpreter).

The reason for the two types has to do with defects in the Intel 386 architecture that were slavishly carried forward into new CPUs for 20 years in the name of backward compatibility. In a nutshell, every CPU with kernel mode and user

mode has a set of instructions that may only be executed in kernel mode, such as instructions that do I/O, change the MMU settings, and so on. In their classic work on virtualization, Popek and Goldberg (1974) called these **sensitive instructions**. There is also a set of instructions that cause a trap if executed in user mode. Popek and Goldberg called these **privileged instructions**. Their paper stated for the first time that a machine is virtualizable only if the sensitive instructions are a subset of the privileged instructions. In simpler language, if you try to do something in user mode that you should not be doing in user mode, the hardware should trap. Unlike the IBM/370, which had this property, the 386 did not. Quite a few sensitive 386 instructions were ignored if executed in user mode. For example, the POPF instruction replaces the flags register, which changes the bit that enables/disables interrupts. In user mode, this bit is simply not changed. As a consequence, the 386 and its successors could not be virtualized, so they could not support a type 1 hypervisor.

Actually, the situation is slightly worse than sketched. In addition to the problems with instructions that fail to trap in user mode, there are instructions that can read sensitive state in user mode without causing a trap. For example, on the Pentium, a program can determine whether it is running in user mode or kernel mode by reading its code segment selector. An operating system that did this and discovered that it was actually in user mode, might make an incorrect decision based on this information.

This problem was solved when Intel and AMD introduced virtualization in their CPUs starting in 2005. On the Intel Core 2 CPUs it is called **VT (Virtualization Technology)**; On the AMD Pacific CPUs it is called **SVM (Secure Virtual Machine)**. We will use the term VT in a generic sense below. Both were inspired by the IBM VM/370 work, but they are slightly different. The basic idea is to create containers in which virtual machines can be run. When a guest operating system is started up in a container, it continues to run there until it causes an exception and traps to the hypervisor, for example, by executing an I/O instruction. The set of operations that trap is controlled by a hardware bitmap set by the hypervisor. With these extensions the classical trap-and-emulate virtual machine approach becomes possible.

8.3.2 Type 1 Hypervisors

Virtualizability is an important issue, so let us examine it a more closely. In Fig. 8-26 we see a type 1 hypervisor supporting one virtual machine. Like all type 1 hypervisors, it runs on the bare metal. The virtual machine runs as a user process in user mode, and as such, is not allowed to execute sensitive instructions. The virtual machine runs a guest operating system that thinks it is in kernel mode, although, of course, it is really in user mode. We will call this **virtual kernel mode**. The virtual machine also runs user processes, which think they are in user mode (and really are in user mode).

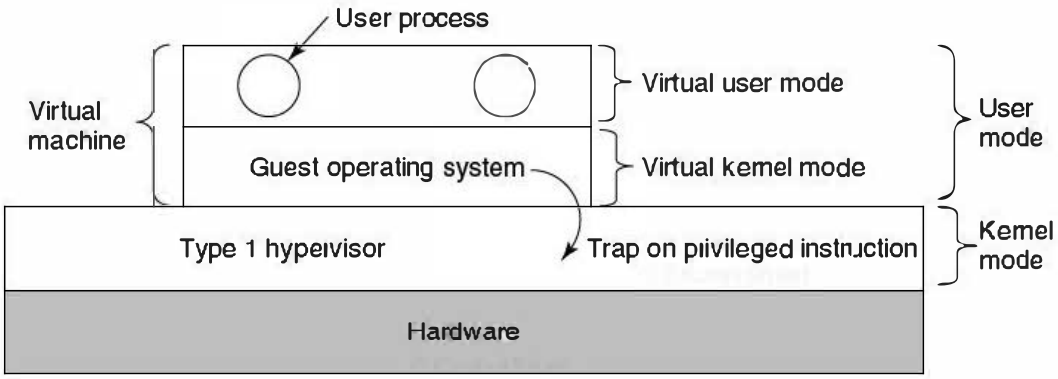

Figure 8-26. When the operating system in a virtual machine executes a kernel-only instruction, it traps to the hypervisor if virtualization technology is present.

What happens when the operating system (which thinks it is kernel mode) executes a sensitive instruction (one allowed only in kernel mode)? On CPUs without VT, the instruction fails and the operating system usually crashes. This makes true virtualization impossible. One could certainly argue that all sensitive instructions should always trap when executed in user mode, but that is not how the 386 and its non-VT successors worked.

On CPUs with VT, when the guest operating system executes a sensitive instruction, a trap to the kernel occurs, as illustrated in Fig. 8-26. The hypervisor can then inspect the instruction to see if it was issued by the guest operating system in the virtual machine or by a user program in the virtual machine. In the former case, it arranges for the instruction to be carried out; in the latter case, it emulates what the real hardware would do when confronted with a sensitive instruction executed in user mode. If the virtual machine does not have VT, the instruction is typically ignored; if it does have VT, it traps to the guest operating system running in the virtual machine.

8.3.3 Type 2 Hypervisors

Building a virtual machine system is relatively straightforward when VT is available, but what did people do before that? Clearly, running a full operating system in a virtual machine would not work because (some of) the sensitive instructions would just be ignored, causing the system to fail. Instead what happened was the invention of what are now called **type 2 hypervisors**, as illustrated in Fig. 1-29(b). The first of these was **VMware** (Adams and Agesen, 2006; and Waldspurger, 2002), which was the outgrowth of the DISCO research project at Stanford University (Bugnion et al., 1997). VMware runs as an ordinary user program on top of a host operating system such as Windows or Linux. When it starts for the first time, it acts like a newly booted computer and expects to find a CD-

ROM containing an operating system in the CD-ROM drive. It then installs the operating system to its **virtual disk** (really just a Windows or Linux file) by running the installation program found on the CD-ROM. Once the guest operating system is installed on the virtual disk, it can be booted at run.

Now let us look at how VMware works in a bit more detail. When executing a Pentium binary program, whether obtained from the installation CD-ROM or from the virtual disk, it scans the code first looking for **basic blocks**, that is, straight runs of instructions ending in a jump, call, trap, or other instruction that changes the flow of control. By definition, no basic block contains any instruction that modifies the program counter except the last one. The basic block is inspected to see if it contains any sensitive instructions (in the Popek and Goldberg sense). If so, each one is replaced with a call to a VMware procedure that handles it. The final instruction is also replaced with a call into VMware.

Once these steps have been taken, the basic block is cached inside VMware and then executed. A basic block not containing any sensitive instructions will execute exactly as fast under VMware as it will on the bare machine—because it is running on the bare machine. Sensitive instructions are caught this way and emulated. This technique is known as **binary translation**.

After the basic block has completed executing, control is returned to VMware, which locates its successor. If the successor has already been translated, it can be executed immediately. If it has not been, it is first translated, cached, then executed. Eventually, most of the program will be in the cache and run at close to full speed. Various optimizations are used, for example, if a basic block ends by jumping to (or calling) another one, the final instruction can be replaced by a jump or call directly to the translated basic block, eliminating all overhead associated with finding the successor block. Also, there is no need to replace sensitive instructions in user programs; the hardware will just ignore them anyway.

It should now be clear why type 2 hypervisors work, even on unvirtualizable hardware: all sensitive instructions are replaced by calls to procedures that emulate these instructions. No sensitive instructions issued by the guest operating system are ever executed by the true hardware. They are turned into calls to the hypervisor, which them emulates them.

One might naively expect that CPUs with VT would greatly outperform the software techniques used by the type 2 hypervisors, but measurements show a mixed picture (Adams and Agesen, 2006). It turns out that the trap-and-emulate approach used by VT hardware generates a lot of traps, and traps are very expensive on modern hardware because they ruin CPU caches, TLBs, and branch prediction tables internal to the CPU. In contrast, when sensitive instructions are replaced by calls to VMware procedures within the executing process, none of this context switching overhead is incurred. As Adams and Agesen show, depending on the workload, sometimes software beats hardware. For this reason, some type 1 hypervisors do binary translation for performance reasons, even though the software will execute correctly without it.

8.3.4 Paravirtualization

Both type 1 and type 2 hypervisors work with unmodified guest operating systems, but have to jump through hoops to get reasonable performance. A different approach that is becoming popular is to modify the source code of the guest operating system so that instead of executing sensitive instructions at all, it makes **hypervisor calls**. In effect the guest operating system is acting like a user program making system calls to the operating system (the hypervisor). When this route is taken, the hypervisor must define an interface consisting of a set of procedure calls that guest operating systems can use. This set of calls forms what is effectively an **API (Application Programming Interface)** even though it is an interface for use by guest operating systems, not application programs.

Going one step further, by removing all the sensitive instructions from the operating system and just having it make hypervisor calls to get system services like I/O, we have turned the hypervisor into a microkernel, like that of Fig. 1-26. A guest operating system from which (some) sensitive instructions have been intentionally removed is said to be **paravirtualized** (Barham et al., 2003; and Whitaker et al., 2002). Emulating peculiar hardware instructions is an unpleasant and time-consuming task. It requires a call into the hypervisor and then emulating the exact semantics of a complicated instruction. It is far better just to have the guest operating system call the hypervisor (or microkernel) to do I/O, and so on. The main reason the first hypervisors just emulated the complete machine was the lack of availability of source code for the guest operating system (e.g., for Windows) or the vast number of variants (e.g., for Linux). Perhaps in the future, the hypervisor/microkernel API will be standardized, and subsequent operating systems will be designed to call it instead of using sensitive instructions. Doing so would make virtual machine technology easier to support and use.

The difference between true virtualization and paravirtualization is illustrated in Fig. 8-27. Here we have two virtual machines being supported on VT hardware. On the left, is an unmodified version of Windows as the guest operating system. When a sensitive instruction is executed, the hardware causes a trap to the hypervisor, which then emulates it and returns. On the right, is a version of Linux modified so that it no longer contains any sensitive instructions. Instead, when it needs to do I/O or change critical internal registers (such as the one pointing to the page tables), it makes a hypervisor call to get the work done, just like an application program making a system call in standard Linux.

In Fig. 8-27 we have shown the hypervisor as being divided into two parts separated by a dashed line. In reality, there is only one program running on the hardware. One part of it is responsible for interpreting trapped sensitive instructions, in this case, from Windows. The other part of it just carries out hypervisor calls. In the figure the latter part is labeled "microkernel." If the hypervisor is intended to run only paravirtualized guest operating systems, there is no need for the emulation of sensitive instructions and we have a true microkernel, which just

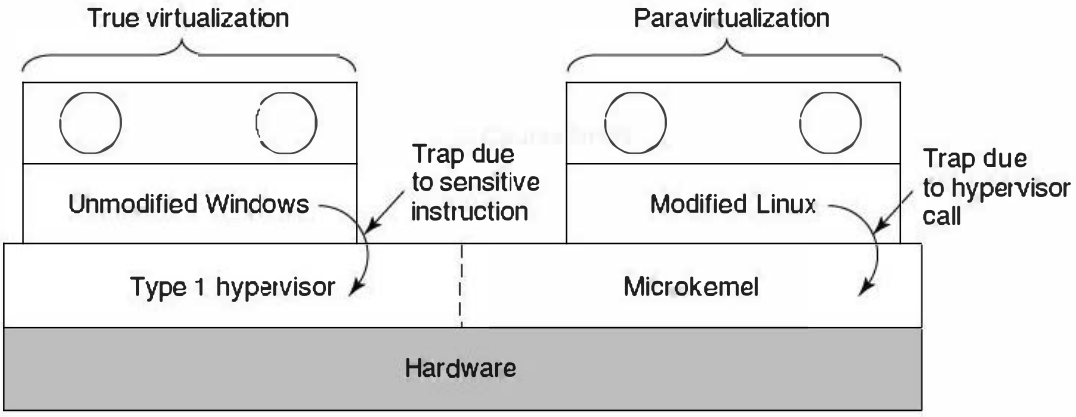

Figure 8-27. A hypervisor supporting both true virtualization and paravirtualization.

provides very basic services such as process dispatching and managing the MMU. The boundary between a type 1 hypervisor and a microkernel is vague already and will get even less clear as hypervisors begin acquiring more and more functionality and hypervisor calls, as seems likely. This subject is controversial, but it is increasingly clear that the program running in kernel mode on the bare hardware should be small and reliable and consist of thousands of lines of code, not millions of lines of code. The topic has been discussed by various researchers (Hand et al., 2005; Heiser et al. 2006; Hohmuth et al., 2004; and Roscoe et al., 2007).

Paravirtualizing the guest operating system raises a number of issues. First, if the sensitive instructions are replaced with calls to the hypervisor, how can the operating system run on the native hardware? After all, the hardware does not understand these hypervisor calls. And second, what if there are multiple hypervisors available in the marketplace, such as VMware, the open-source Xen originally from the University of Cambridge, and Microsoft's Viridian, all with somewhat different hypervisor APIs? How can the kernel be modified to run on all of them?

Amsden et al. (2006) have proposed a solution. In their model, the kernel is modified to call special procedures whenever it needs to do something sensitive. Together these procedures, called the **VMI (Virtual Machine Interface)** form a low-level layer that interfaces with the hardware or hypervisor. These procedures are designed to be generic and not tied to the hardware or to any particular hypervisor.

An example of this technique is given in Fig. 8-28 for a paravirtualized version of Linux they call VMI Linux (VMIL). When VMI Linix runs on the bare hardware, it has to be linked with a library that issues the actual (sensitive) instruction needed to do the work, as shown in Fig. 8-28(a). When running on a hypervisor, say VMware or Xen, the guest operating system is linked with different libraries that make the appropriate (and different) hypervisor calls to the

underlying hypervisor. In this way, the core of the operating system remains portable yet is hypervisor friendly and still efficient.

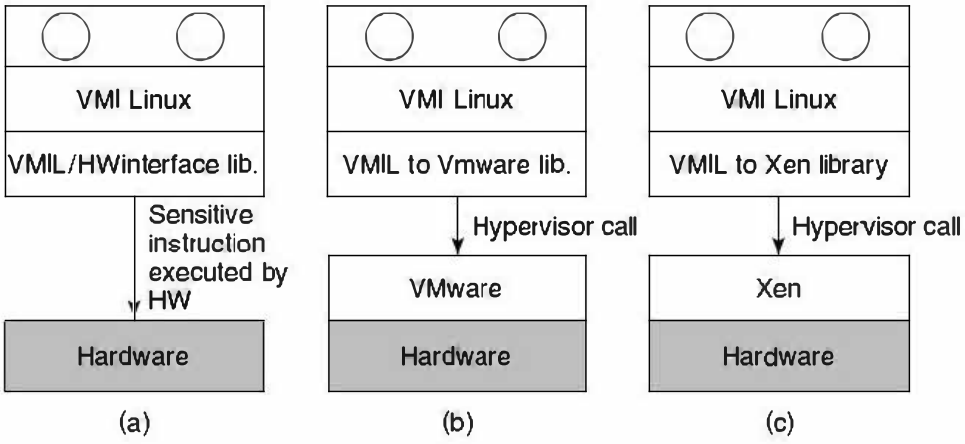

Figure 8-28. VMI Linux running on (a) the bare hardware (b) VMware (c) Xen.

Other proposals for a virtual machine interface have also been made. Another popular one is called **paravirt ops**. The idea is conceptually similar to what we described above, but different in the details.

8.3.5 Memory Virtualization

So far we have just addressed the issue of how to virtualize the CPU. But a computer system has more than just a CPU. It also has memory and I/O devices. They have to be virtualized, too. Let us see how that is done.

Modern operating systems nearly all support virtual memory, which is basically a mapping of pages in the virtual address space onto pages of physical memory. This mapping is defined by (multilevel) page tables. Typically the mapping is set in motion by having the operating system set a control register in the CPU that points to the top-level page table. Virtualization greatly complicates memory management.

Suppose, for example, a virtual machine is running, and the guest operating system in it decides to map its virtual pages 7, 4, and 3 onto physical pages 10, 11, and 12, respectively. It builds page tables containing this mapping and loads a hardware register to point to the top-level page table. This instruction is sensitive. On a VT CPU, it will trap; with VMware it will cause a call to a VMware procedure; on a paravirtualized operating system, it will generate a hypervisor call. For simplicity, let us assume it traps into a type 1 hypervisor, but the problem is the same in all three cases.

What does the hypervisor do now? One solution is to actually allocate physical pages 10, 11, and 12 to this virtual machine and set up the actual page tables to map the virtual machine's virtual pages 7, 4, and 3 to use them. So far, so good.

Now suppose a second virtual machine starts and maps its virtual pages 4, 5, and 6 onto physical pages 10, 11, and 12 and loads the control register to point to its page tables. The hypervisor catches the trap, but what should it do? It cannot use this mapping because physical pages 10, 11, and 12 are already in use. It can find some free pages, say 20, 21, and 22 and use them, but it first has to create new page tables mapping the virtual pages 4, 5, and 6 of virtual machine 2 onto 20, 21, and 22. If another virtual machine starts and tries to use physical pages 10, 11, and 12, it has to create a mapping for it. In general, for each virtual machine the hypervisor needs to create a **shadow page table** that maps the virtual pages used by the virtual machine onto the actual pages the hypervisor gave it.

Worse yet, every time the guest operating system changes its page tables, the hypervisor must change the shadow page tables as well. For example, if the guest OS remaps virtual page 7 onto what it sees as physical page 200 (instead of 10), the hypervisor has to know about this change. The trouble is that the guest operating system can change its page tables by just writing to memory. No sensitive operations are required, so the hypervisor does not even know about the change and certainly cannot update the shadow page tables used by the actual hardware.

A possible (but clumsy) solution, is for the hypervisor to keep track of which page in the guest's virtual memory contains the top-level page table. It can get this information the first time the guest attempts to load the hardware register that points to it because this instruction is sensitive and traps. The hypervisor can create a shadow page table at this point and also map the top-level page table and the page tables it points to as read only. Subsequent attempts by the guest operating system to modify any of them will cause a page fault and thus give control to the hypervisor, which can analyze the instruction stream, figure out what the guest OS is trying to do, and update the shadow page tables accordingly. It is not pretty, but it is doable in principle.

This is an area in which future versions of VT could provide assistance by doing a two-level mapping in hardware. The hardware could first map the virtual page to the guest's idea of the physical page, then map that address (which the hardware sees as a virtual address) onto the physical address, all without causing any traps. In this way no page tables would have to be marked as read only and the hypervisor would merely have to provide a mapping between each guest's virtual address space and physical memory. When switching virtual machines, it would just change this mapping, the same way a normal operating system changes the mapping when switching processes.

In a paravirtualized operating system, the situation is different. Here the paravirtualized OS in the guest knows that when it is finished changing some process' page table, it had better inform the hypervisor. Consequently, it first changes the page table completely, then issues a hypervisor call telling the hypervisor about the new page table. Thus instead of getting a protection fault on every update to the page table, there is one hypervisor call when the whole thing has been updated, obviously a more efficient way to do business.

8.3.6 I/O Virtualization

Having looked at CPU and memory virtualization, the next step is to examine I/O virtualization. The guest operating system typically will start out probing the hardware to find out what kinds of I/O devices are attached. These probes will trap to the hypervisor. What should the hypervisor do? One approach is for it to report back that the disks, printers, and so on are the ones that the hardware actually has. The guest will then load device drivers for these devices and try to use them. When the device drivers try to do actual I/O, they will read and write the device's hardware device registers. These instructions are sensitive and will trap to the hypervisor, which could then copy the needed values to and from the hardware registers, as needed.

But here, too, we have a problem. Each guest OS thinks it owns an entire disk partition, and there may be many more virtual machines (hundreds) than there are disk partitions. The usual solution is for the hypervisor to create a file or region on the actual disk for each virtual machine's physical disk. Since the guest OS is trying to control a disk that the real hardware has (and which the hypervisor understands), it can convert the block number being accessed into an offset into the file or disk region being used for storage and do the I/O.

It is also possible for the disk that the guest is using to be different from the real one. For example, if the actual disk is some brand-new high-performance disk (or RAID) with a new interface, the hypervisor could advertise to the guest OS that it has a plain old IDE disk and let the guest OS install an IDE disk driver. When this driver issues IDE disk commands, the hypervisor converts them into commands to drive the new disk. This strategy can be used to upgrade the hardware without changing the software. In fact, this ability of virtual machines to remap hardware devices was one of the reasons VM/370 became popular: companies wanted to buy new and faster hardware but did not want to change their software. Virtual machine technology made this possible.

Another I/O problem that must be solved somehow is the use of DMA, which uses absolute memory addresses. As might be expected, the hypervisor has to intervene here and remap the addresses before the DMA starts. However, hardware is starting to appear with an **I/O MMU**, which virtualizes the I/O the same way the MMU virtualizes the memory. This hardware eliminates the DMA problem.

A different approach to handling I/O is to dedicate one of the virtual machines to running a standard operating system and reflect all I/O calls from the other ones to it. This approach is enhanced when paravirtualization is used, so the command being issued to the hypervisor actually says what the guest OS wants (e.g., read block 1403 from disk 1) rather than being a series of commands writing to device registers, in which case the hypervisor has to play Sherlock Holmes and figure out what it is trying to do. Xen uses this approach to I/O, with the virtual machine that does I/O called **domain 0**.

I/O virtualization is an area in which type 2 hypervisors have a practical advantage over type 1 hypervisors: the host operating system contains the device drivers for all the weird and wonderful I/O devices attached to the computer. When an application program attempts to access a strange I/O device, the translated code can call the existing device driver to get the work done. With a type 1 hypervisor, the hypervisor must either contain the driver itself, or make a call to a driver in domain 0, which is somewhat similar to a host operating system. As virtual machine technology matures, future hardware is likely to allow application programs to access the hardware directly in a secure way, meaning that device drivers can be linked directly with application code or put in separate user-mode servers, thereby eliminating the problem.

8.3.7 Virtual Appliances

Virtual machines offer an interesting solution to a problem that has long plagued users, especially users of open-source software: how to install new application programs. The problem is that many applications are dependent on numerous other applications and libraries, which are themselves dependent on a host of other software packages, and so on. Furthermore, there may be dependencies on particular versions of the compilers, scripting languages, and the operating system.

With virtual machines now available, a software developer can carefully construct a virtual machine, load it with the required operating system, compilers, libraries, and application code, and freeze the entire unit, ready to run. This virtual machine image can then be put on a CD-ROM or a Website for customers to install or download. This approach means that only the software developer has to understand all the dependencies. The customers get a complete package that actually works, completely independent of which operating system they are running and which other software, packages, and libraries they have installed. These "shrink-wrapped" virtual machines are often called **virtual appliances**.

8.3.8 Virtual Machines on Multicore CPUs

The combination of virtual machines and multicore CPUs opens a whole new world in which the number of CPUs available can be set in software. If there are, say, four cores, and each one can be used to run, for example, up to eight virtual machines, a single (desktop) CPU can be configured as a 32-node multicomputer if need be, but it can also have fewer CPUs, depending on the needs of the software. Never before has it been possible for an application designer to first choose how many CPUs he wants and then write the software accordingly. This clearly represents a new phase in computing.

Although it is not so common yet, it is certainly conceivable that virtual machines could share memory. All that has to be done is map physical pages into the

address spaces of multiple virtual machines. If this can be done, a single computer becomes a virtual multiprocessor. Since all the cores in a multicore chip share the same RAM, a single quad-core chip could easily be configured as a 32-node multiprocessor or a 32-node multicomputer. as needed.

The combination of multicore, virtual machines, and hypervisors and microkernels is going to radically change the way people think about computer systems. Current software cannot deal with the idea of the programmer determining how many CPUs are needed, whether they should be set up as a multicomputer or a multiprocessor, and how minimal kernels of one kind or another fit into the picture. Future software will have to deal with these issues.

8.3.9 Licensing Issues

Most software is licensed on a per-CPU basis. In other words, when you buy a program, you have the right to run it on just one CPU. Does this contract give you the right to run the software on multiple virtual machines all running on the same physical machine? Many software vendors are somewhat unsure of what to do here.

The problem is much worse in companies that have a license allowing them to have n machines running the software at the same time, especially when virtual machines come and go on demand.

In some cases, software vendors have put an explicit clause in the license forbidding the licensee from running the software on a virtual machine or on an unauthorized virtual machine. Whether any of these restrictions will hold up in court and how users respond to them remains to be seen.

8.4 DISTRIBUTED SYSTEMS

Having now completed our study of multiprocessors, multicomputers, and virtual machines, it is time to turn to the last type of multiple processor system, the **distributed system**. These systems are similar to multicomputers in that each node has its own private memory, with no shared physical memory in the system. However, distributed systems are even more loosely coupled than multicomputers.

To start with, the nodes of a multicomputer generally have a CPU, RAM, a network interface, and perhaps a hard disk for paging. In contrast, each node in a distributed system is a complete computer, with a full complement of peripherals. Next, the nodes of a multicomputer are normally in a single room, so they can communicate by a dedicated high-speed network, whereas the nodes of a distributed system may be spread around the world. Finally, all the nodes of a multicomputer run the same operating system, share a single file system, and are under a common administration, whereas the nodes of a distributed system may each run a

directory. Eventually, mainframes developed complex hierarchical file systems, perhaps culminating in the MULTICS file system.

As minicomputers came into use, they eventually also had hard disks. The standard disk on the PDP-11 when it was introduced in 1970 was the RK05 disk, with a capacity of 2.5 MB, about half of the IBM RAMAC, but it was only about 40 cm in diameter and 5 cm high. But it, too, had a single-level directory initially. When microcomputers came out, CP/M was initially the dominant operating system, and it, too, supported just one directory on the (floppy) disk.

Virtual Memory

Virtual memory (discussed in Chap. 3), gives the ability to run programs larger than the machine's physical memory by moving pieces back and forth between RAM and disk. It underwent a similar development, first appearing on mainframes, then moving to the minis and the micros. Virtual memory also enabled the ability to have a program dynamically link in a library at run time instead of having it compiled in. MULTICS was the first system to allow this. Eventually, the idea propagated down the line and is now widely used on most UNIX and Windows systems.

In all these developments, we see ideas that are invented in one context and later thrown out when the context changes (assembly language programming, monoprogramming, single-level directories, etc.) only to reappear in a different context often a decade later. For this reason in this book we will sometimes look at ideas and algorithms that may seem dated on today's gigabyte PCs, but which may soon come back on embedded computers and smart cards.

1.6 SYSTEM CALLS

We have seen that operating systems have two main functions: providing abstractions to user programs and managing the computer's resources. For the most part, the interaction between user programs and the operating system deals with the former; for example, creating, writing, reading, and deleting files. The resource management part is largely transparent to the users and done automatically. Thus the interface between user programs and the operating system is primarily about dealing with the abstractions. To really understand what operating systems do, we must examine this interface closely. The system calls available in the interface vary from operating system to operating system (although the underlying concepts tend to be similar).

We are thus forced to make a choice between (1) vague generalities ("operating systems have system calls for reading files") and (2) some specific system ("UNIX has a read system call with three parameters: one to specify the file, one to tell where the data are to be put, and one to tell how many bytes to read").

We have chosen the latter approach. It's more work that way, but it gives more insight into what operating systems really do. Although this discussion specifically refers to POSIX (International Standard 9945-1), hence also to UNIX, System V, BSD, Linux, MINIX 3, and so on, most other modern operating systems have system calls that perform the same functions, even if the details differ. Since the actual mechanics of issuing a system call are highly machine dependent and often must be expressed in assembly code, a procedure library is provided to make it possible to make system calls from C programs and often from other languages as well.

It is useful to keep the following in mind. Any single-CPU computer can execute only one instruction at a time. If a process is running a user program in user mode and needs a system service, such as reading data from a file, it has to execute a trap instruction to transfer control to the operating system. The operating system then figures out what the calling process wants by inspecting the parameters. Then it carries out the system call and returns control to the instruction following the system call. In a sense, making a system call is like making a special kind of procedure call, only system calls enter the kernel and procedure calls do not.

To make the system call mechanism clearer, let us take a quick look at the read system call. As mentioned above, it has three parameters: the first one specifying the file, the second one pointing to the buffer, and the third one giving the number of bytes to read. Like nearly all system calls, it is invoked from C programs by calling a library procedure with the same name as the system call: *read*. A call from a C program might look like this:

```
count = read(fd, buffer, nbytes);
```

The system call (and the library procedure) return the number of bytes actually read in *count*. This value is normally the same as *nbytes*, but may be smaller, if, for example, end-of-file is encountered while reading.

If the system call cannot be carried out, either due to an invalid parameter or a disk error, *count* is set to −1, and the error number is put in a global variable, *errno*. Programs should always check the results of a system call to see if an error occurred.

System calls are performed in a series of steps. To make this concept clearer, let us examine the read call discussed above. In preparation for calling the *read* library procedure, which actually makes the read system call, the calling program first pushes the parameters onto the stack, as shown in steps 1-3 in Fig. 1-17.

C and C++ compilers push the parameters onto the stack in reverse order for historical reasons (having to do with making the first parameter to *printf*, the format string, appear on top of the stack). The first and third parameters are called by value, but the second parameter is passed by reference, meaning that the address of the buffer (indicated by &) is passed, not the contents of the buffer. Then

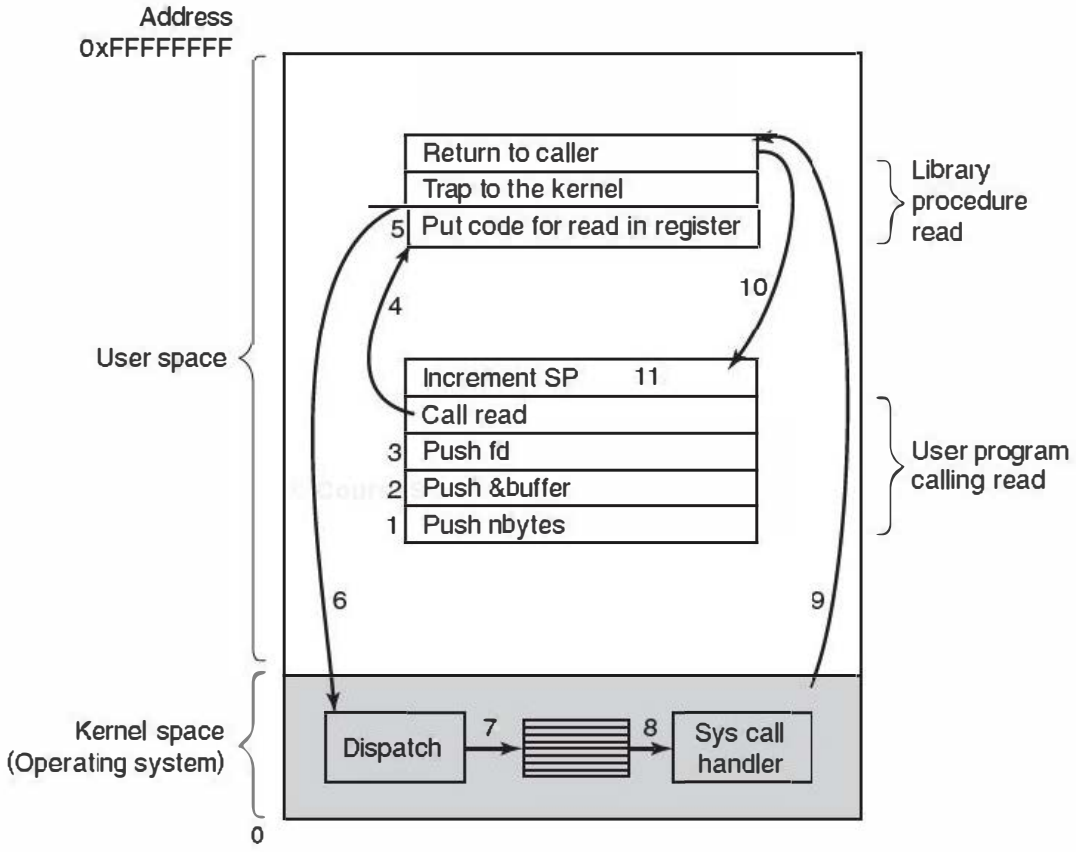

Figure 1-17. The 11 steps in making the system call read(fd, buffer, nbytes).

comes the actual call to the library procedure (step 4). This instruction is the normal procedure call instruction used to call all procedures.

The library procedure, possibly written in assembly language, typically puts the system call number in a place where the operating system expects it, such as a register (step 5). Then it executes a TRAP instruction to switch from user mode to kernel mode and start execution at a fixed address within the kernel (step 6). The TRAP instruction is actually fairly similar to the procedure call instruction in the sense that the instruction following it is taken from a distant location and the return address is saved on the stack for use later.

Nevertheless, the TRAP instruction also differs from the procedure call instruction in two fundamental ways. First, as a side effect, it switches into kernel mode. The procedure call instruction does not change the mode. Second, rather than giving a relative or absolute address where the procedure is located, the TRAP instruction cannot jump to an arbitrary address. Depending on the architecture, it either jumps to a single fixed location, there is an 8-bit field in the instruction giving the index into a table in memory containing jump addresses, or equivalent.

The kernel code that starts following the TRAP examines the system call number and then dispatches to the correct system call handler, usually via a table of

pointers to system call handlers indexed on system call number (step 7). At that point the system call handler runs (step 8). Once the system call handler has completed its work, control may be returned to the user-space library procedure at the instruction following the TRAP instruction (step 9). This procedure then returns to the user program in the usual way procedure calls return (step 10).

To finish the job, the user program has to clean up the stack, as it does after any procedure call (step 11). Assuming the stack grows downward, as it often does, the compiled code increments the stack pointer exactly enough to remove the parameters pushed before the call to *read*. The program is now free to do whatever it wants to do next.

In step 9 above, we said "may be returned to the user-space library procedure" for good reason. The system call may block the caller, preventing it from continuing. For example, if it is trying to read from the keyboard and nothing has been typed yet, the caller has to be blocked. In this case, the operating system will look around to see if some other process can be run next. Later, when the desired input is available, this process will get the attention of the system and steps 9–11 will occur.

In the following sections, we will examine some of the most heavily used POSIX system calls, or more specifically, the library procedures that make those system calls. POSIX has about 100 procedure calls. Some of the most important ones are listed in Fig. 1-18, grouped for convenience in four categories. In the text we will briefly examine each call to see what it does.

To a large extent, the services offered by these calls determine most of what the operating system has to do, since the resource management on personal computers is minimal (at least compared to big machines with multiple users). The services include things like creating and terminating processes, creating, deleting, reading, and writing files, managing directories, and performing input and output.

As an aside, it is worth pointing out that the mapping of POSIX procedure calls onto system calls is not one-to-one. The POSIX standard specifies a number of procedures that a conformant system must supply, but it does not specify whether they are system calls, library calls, or something else. If a procedure can be carried out without invoking a system call (i.e., without trapping to the kernel), it will usually be done in user space for reasons of performance. However, most of the POSIX procedures do invoke system calls, usually with one procedure mapping directly onto one system call. In a few cases, especially where several required procedures are only minor variations of one another, one system call handles more than one library call.

1.6.1 System Calls for Process Management

The first group of calls in Fig. 1-18 deals with process management. Fork is a good place to start the discussion. Fork is the only way to create a new process in POSIX. It creates an exact duplicate of the original process, including all the file