which then locks the scheduling queues and selects the highest priority process, *A*, as shown in Fig. 8-12(b). Next, CPU 12 goes idle and chooses process *B*, as illustrated in Fig. 8-12(c). As long as the processes are completely unrelated, doing scheduling this way is a reasonable choice.

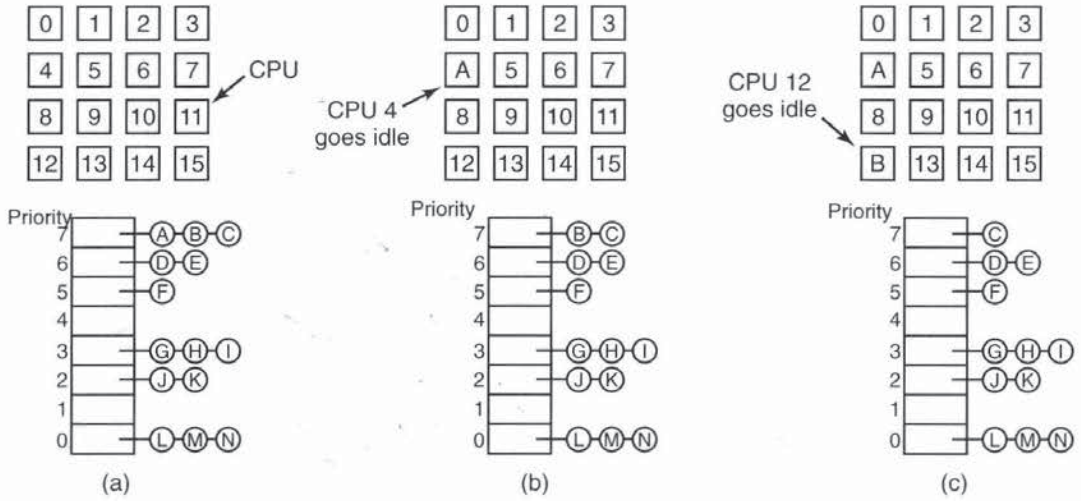


Figure 8-12. Using a single data structure for scheduling a multiprocessor.

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 numbers of CPUs grows and the usual overhead in doing a context switch when a process blocks for I/O.

It is also possible that a context switch happens when a process' quantum expires. On a multiprocessor, that has certain properties not present on a uniprocessor. Suppose that the process holds a spin lock, not unusual on multiprocessors, as discussed above. Other CPUs waiting on the spin lock just waste their time spinning until that process 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 process 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 process acquiring a spin lock sets a process-wide 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 process 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 process *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 process' 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 process 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 process is created, it is assigned to a CPU, for example based on which one has the smallest load at that moment. This assignment of processes to CPUs is the top level of the algorithm. As a result, each CPU acquires its own collection of processes.

The actual scheduling of the processes 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 process on the same CPU, cache affinity is maximized. However, if a CPU has no processes 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 processes 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 creates multiple threads that work together. For our purposes, a job consisting of multiple related processes or a process consisting of multiple kernel threads are essentially the same thing. We will refer to the schedulable entities as threads here, but the material holds for processes as well. 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 processes come and go.

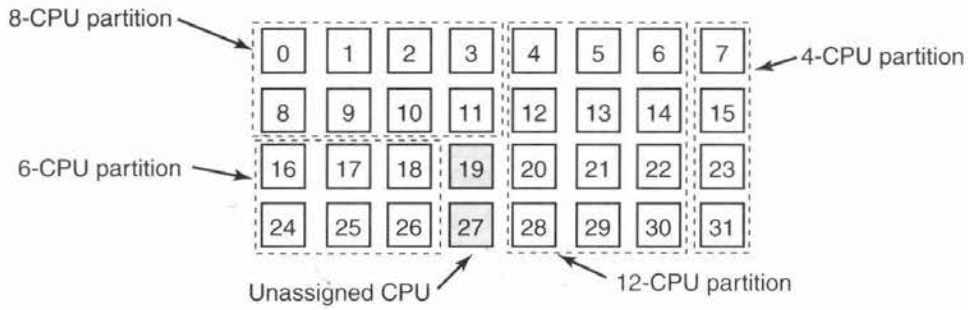


Figure 8-13. A set of 32 CPUs split into four partitions, with two CPUs available.

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 process 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 process 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 processes to actively manage the degree of parallelism. One way to do manage the parallelism is to have a central server that keeps track of which processes are running and want to run and what their minimum and maximum CPU requirements are (Tucker and Gupta, 1989). Periodically, each CPU polls the central server to ask how many CPUs it may use. It then adjusts the number of processes or threads up or down to match what is available. For example, a Web server can have 1, 2, 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 5, when the next 5 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 processes 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 (or processes of a job) 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. Suppose that luck has it that A_0 and B_1 start first, as shown in Fig. 8-14.

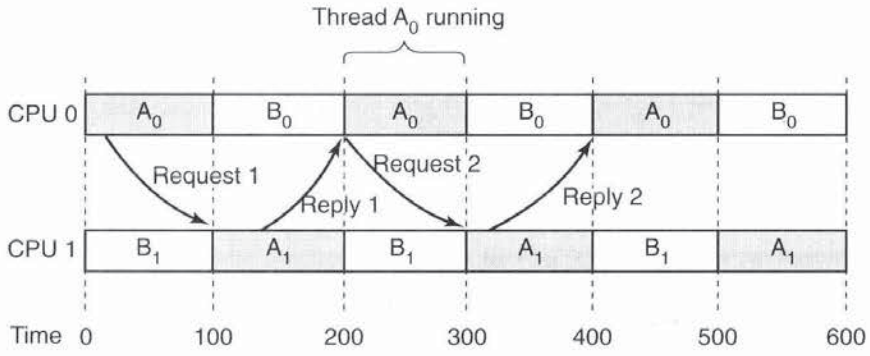


Figure 8-14. Communication between two threads belonging to process A that are running out of phase.

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.

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 simultaneously, 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. 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 following 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 process E run in time slot 3. Then the cycle repeats, with slot 4 being the same as slot 0 and so on.

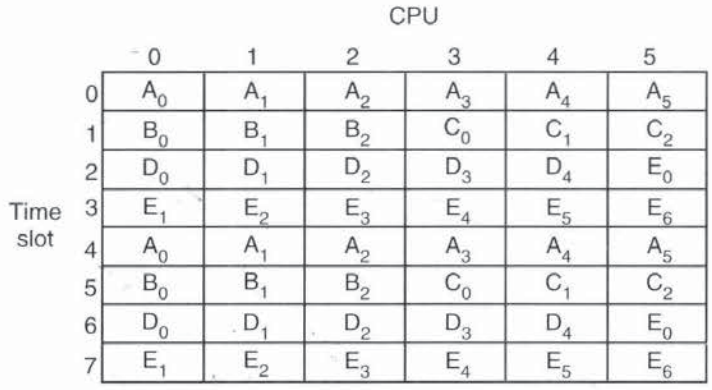


Figure 8-15. Gang scheduling.

The idea of gang scheduling is to have all the threads of a process run together, 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.

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.

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**).

Multicomputers are easy to build because the basic component is just a stripped-down PC with the addition of a 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. 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 graphics adapter, monitor, keyboard, and mouse are nearly always absent. In some cases, the PC contains a 2-way or 4-way multiprocessor board 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 to either 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.

As an alternative to the single switch design, the nodes may form a ring, with two wires coming out the network interface card, one going 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 general case it could be a $k \times k \times k$ cube. In Fig. 8-16(f) we have a four-dimensional cube constructed from two three-dimensional cubes with the corresponding nodes connected. We could make a

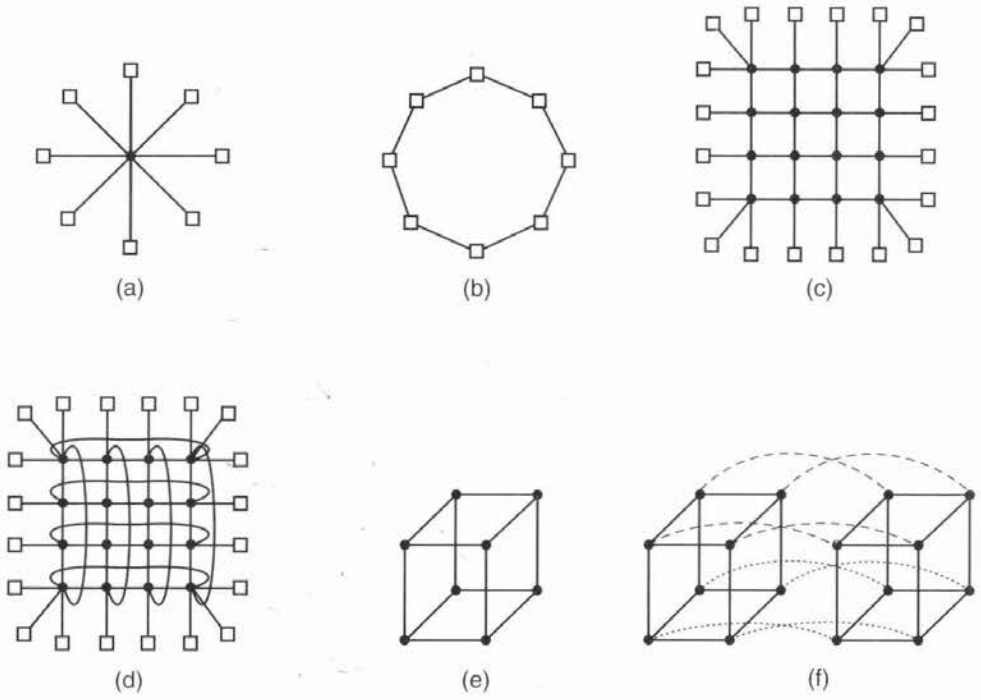


Figure 8-16. Various interconnect topologies. (a) A single switch. (b) A ring. (c) A grid. (d) A double torus. (e) A cube. (f) A 4D hypercube.

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 this 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, so, 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 has 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, it is copied 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.

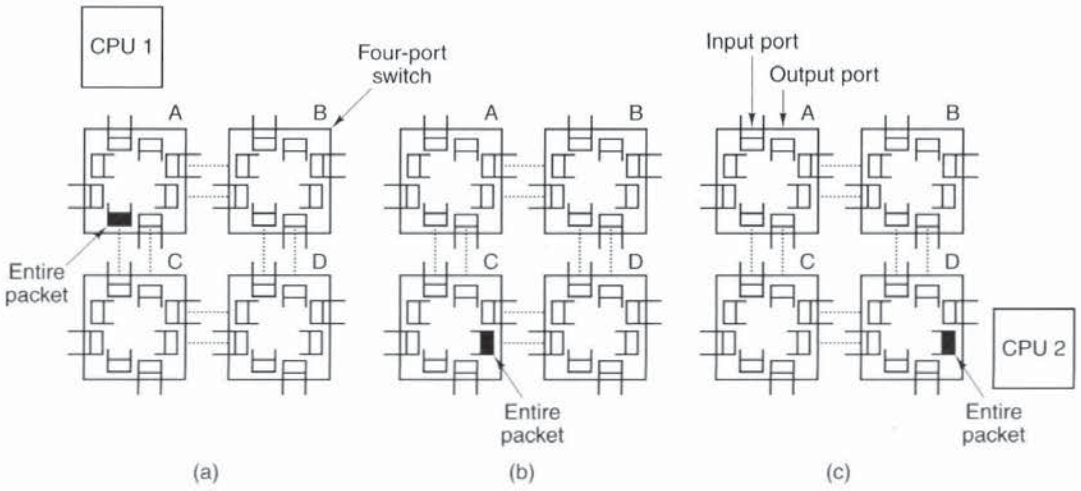


Figure 8-17. Store-and-forward packet switching.

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 design a hybrid network, with some of the properties of circuit switching and some of the properties of packet switching. For example, each packet can be logically divided into smaller units. As soon as the first unit arrives at a switch, it can be moved to the next switch, even before the tail of the packet has arrived.

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. 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 (Bhoedjang, 2000). Other references are (Buzzard et al., 1996; Pakin et al., 1997; Steenkiste, 1994; and Von Eicken et al., 1992).

In virtually all multicomputers, the interface board contains some 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.

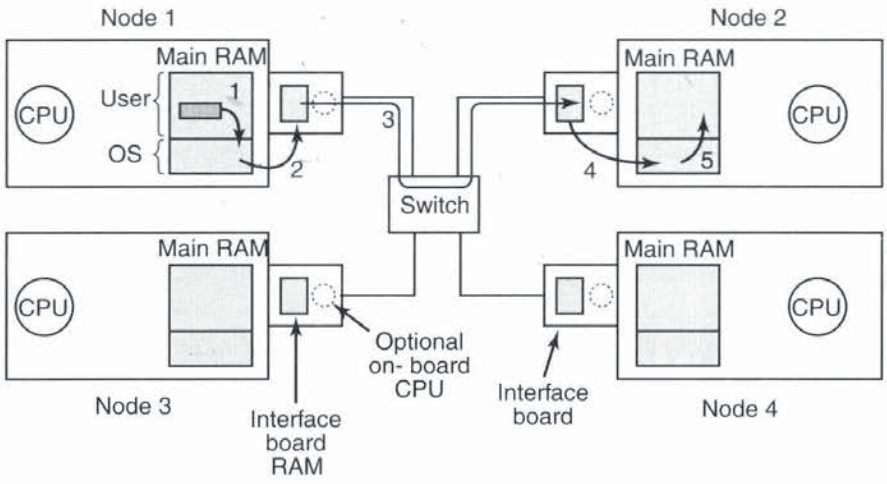


Figure 8-18. Position of the network interface boards in a multicomputer.

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 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.

Some interface boards have a full CPU on them, possibly in addition to one or more DMA channels. 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), and taking care of protection in a system with 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.

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 only send a packet 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 dominate the performance, the extra copies to and from the kernel may double the end-to-end delay and cut the bandwidth 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, only work when the processes are assumed to be cooperating. In a timesharing 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 only really works well when there is just one user process running on each node unless special precautions are taken (for example, 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, even on a timesharing basis. 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.

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 uses physical rather than virtual addresses and runs independently of the CPU. 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.

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 (Bhoedjang, 2000).

In theory, the same problem occurs with DMA from a disk or other device, but since these are set up by the operating system to kernel buffers, it is easy for

the system to avoid paging the buffers. The problem here is that the user is setting up and managing the DMA, and the operating system does not know that removing a page could be fatal, something it does know for I/O that it starts itself. The reason using kernel buffers is acceptable for disk I/O and not for multiprocessor communication is that an extra 20-μsec delay is tolerable for disk latency but not for process-to-process communication latency.

The DMA problem can be avoided by having the user process first pin one page at startup and asking for its physical address. Outgoing packets are first copied there and then to the network interface, but this extra copy is just as bad as copying to the kernel.

For these reasons, using programmed I/O to and from the interface board is usually the safest course, since any page faults encountered are just ordinary CPU page faults and can be handled in the usual way by the operating system. When a page fault occurs, the copy loop stops instantly and remains stopped until the operating system has handled the page fault. A more sophisticated scheme is to use programmed I/O for small packets and DMA with pinning and unpinning for large ones.

If the network interface boards have their own CPUs (e.g., as do Myrinet boards), these on-board CPUs can be used to speed up communication. However, care has to be taken to avoid race conditions between the main CPU and the on-board CPU. One way to avoid races is illustrated in Fig. 8-19, where we focus on node 1 sending packets and node 2 receiving them, not necessarily from each other. The key synchronization data structure for senders is the send ring; for receivers it is the receive ring. All nodes have both since they all send and receive. Each ring has room for n packets. There is also a bitmap per ring with n bits, possibly separate (as shown) or possibly integrated into the rings, telling which ring slots are currently valid.

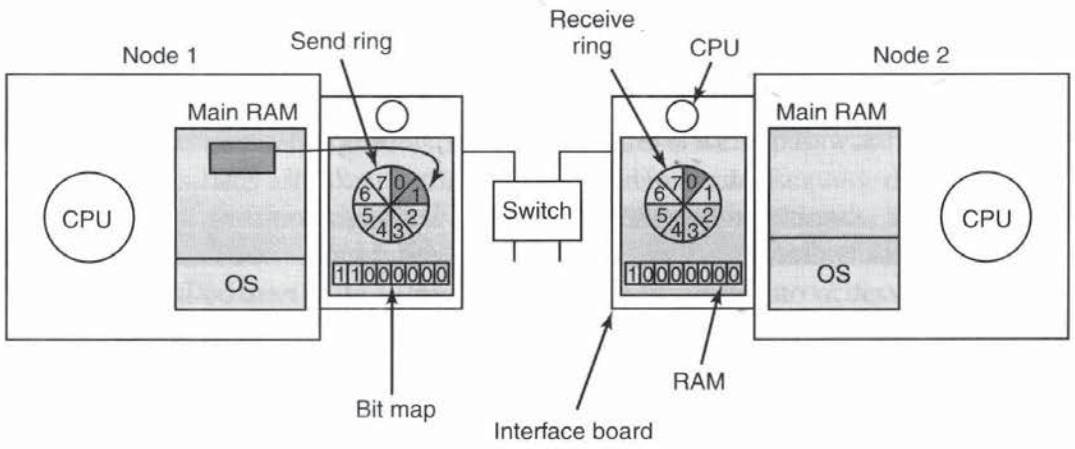


Figure 8-19. Use of send and receive rings to coordinate the main CPU with the on-board CPU.

When a sender has a new packet to send, it first checks to see if there is an available slot in the send ring. If not, it must wait, to prevent overrun. If there is a slot, it copies the packet to the next available slot, and after that is completed, sets the corresponding bit in the bitmap. When the on-board CPU has finished whatever it is doing, it checks the send ring. If it contains any packets, it takes the one there longest and transmits it. When it is done, it clears the corresponding bit in the bitmap. Since the main CPU is the only one that sets the bits and the on-board CPU is the only one that clears them, there are no race conditions. The receive ring works the other way, with the on-board CPU setting a bit to indicate packet arrival and the main CPU clearing it to indicate that it has copied the packet and freed the buffer.

This scheme can also be used even without programmed I/O done by the main CPU. In that case, the send ring entry does not contain the packet itself, but a pointer to the packet in the main RAM. When the on-board CPU is ready to transmit the packet, it fetches the packet to the interface board, either using programmed I/O itself or via DMA. In both cases, this approach works only if the page containing the packet is known to be pinned.

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 calling 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 caller to be blocked until the message has been sent. The latter causes the caller to be blocked until a message arrives. When one does, the message is copied to the buffer pointed to by *mptr* and the caller 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. 8-20(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.

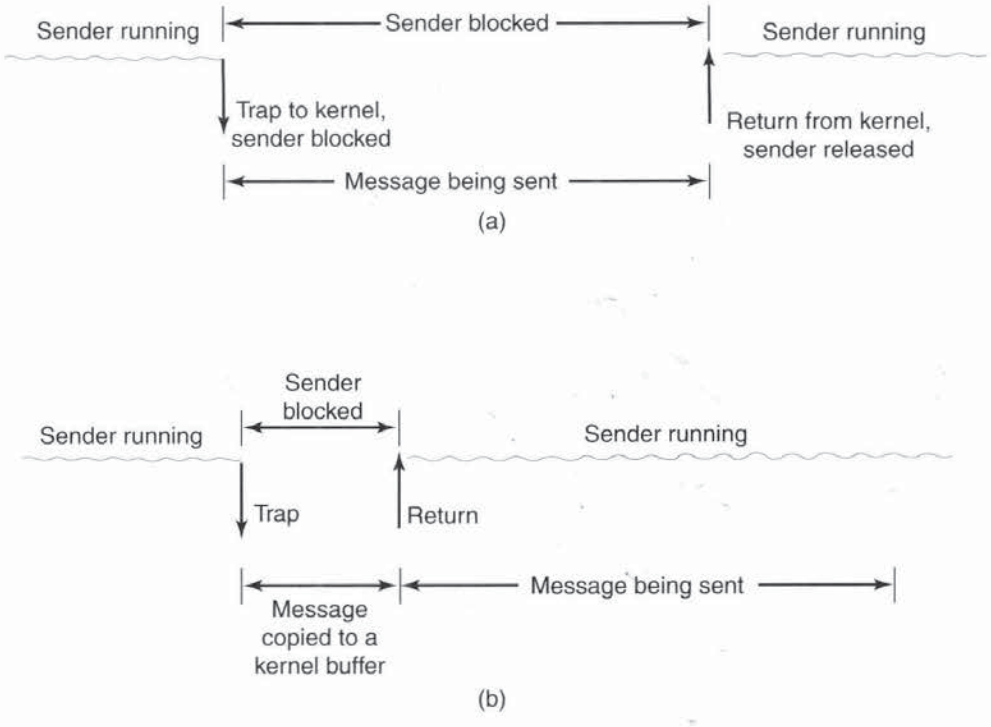


Figure 8-20. (a) A blocking send call. (b) A nonblocking send call.

An alternative to blocking calls are **nonblocking calls** (sometimes called **asynchronous calls**). If *send* is nonblocking, it returns control to the caller 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. 8-20(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 the sender when the message has been 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 best one, 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. 8-20(a) to Fig. 8-20(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 caller 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 caller 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 only work 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 caller 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 a RPC are shown in Fig. 8-21. 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 **marshaling**. 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.

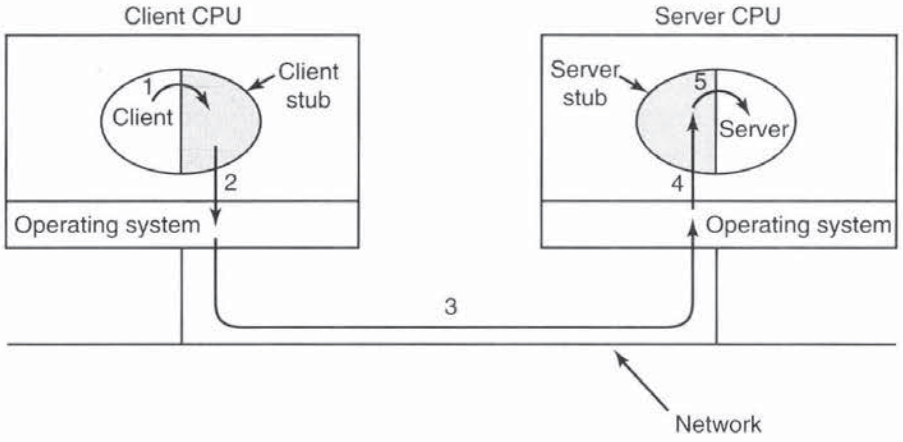


Figure 8-21. Steps in making a remote procedure call. The stubs are shaded gray.

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 procedure. 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 procedure may communicate using global variables, in addition to 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). With DSM, each page is located in one of the memories of Fig. 8-1. Each machine has its own virtual memory and its own 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 is 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. 8-22. In Fig. 8-22(a), we see a true multiprocessor with physical shared memory implemented by the hardware. In Fig. 8-22(b), we see DSM, implemented by the operating system. In Fig. 8-22(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.

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. 8-23(a) for an address space with 16 pages and four nodes, each capable of holding four pages.

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. 8-23(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. 8-23

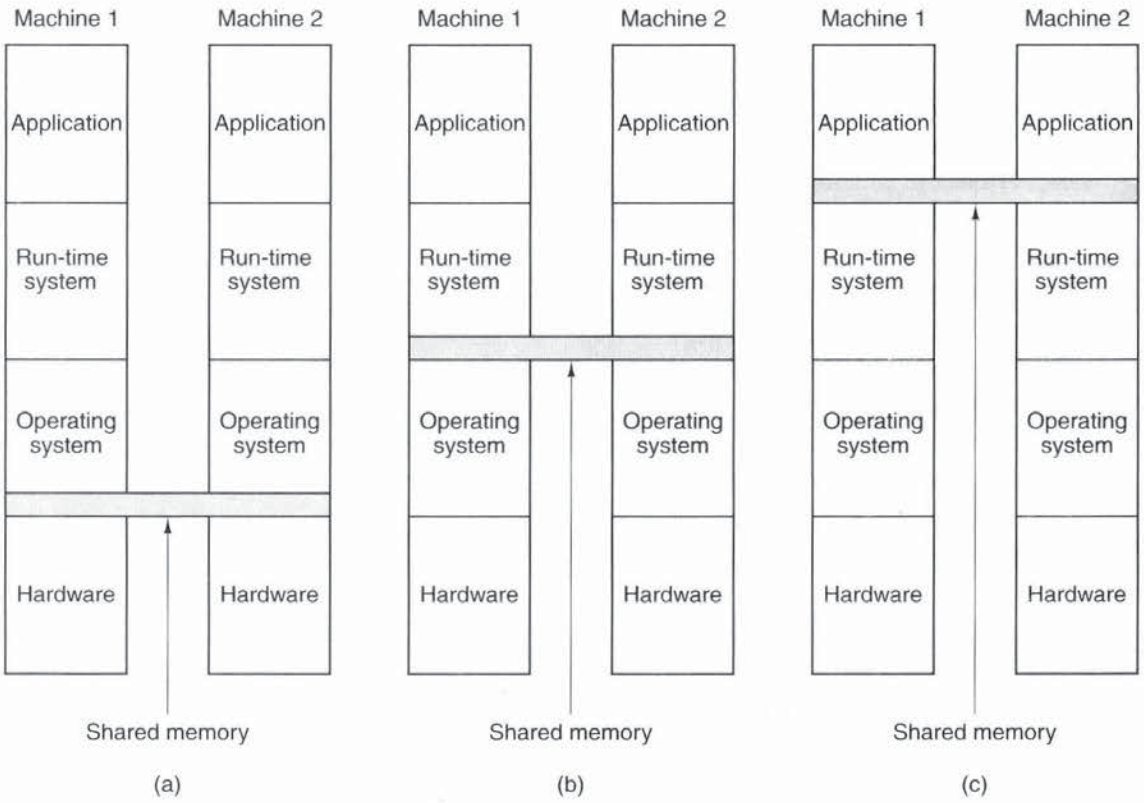


Figure 8-22. Various layers where shared memory can be implemented. (a) The hardware. (b) The operating system. (c) User-level software.

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 disturbed, as shown in Fig. 8-23(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 should the chunk be? In multiprocessors, the cache block size is usually

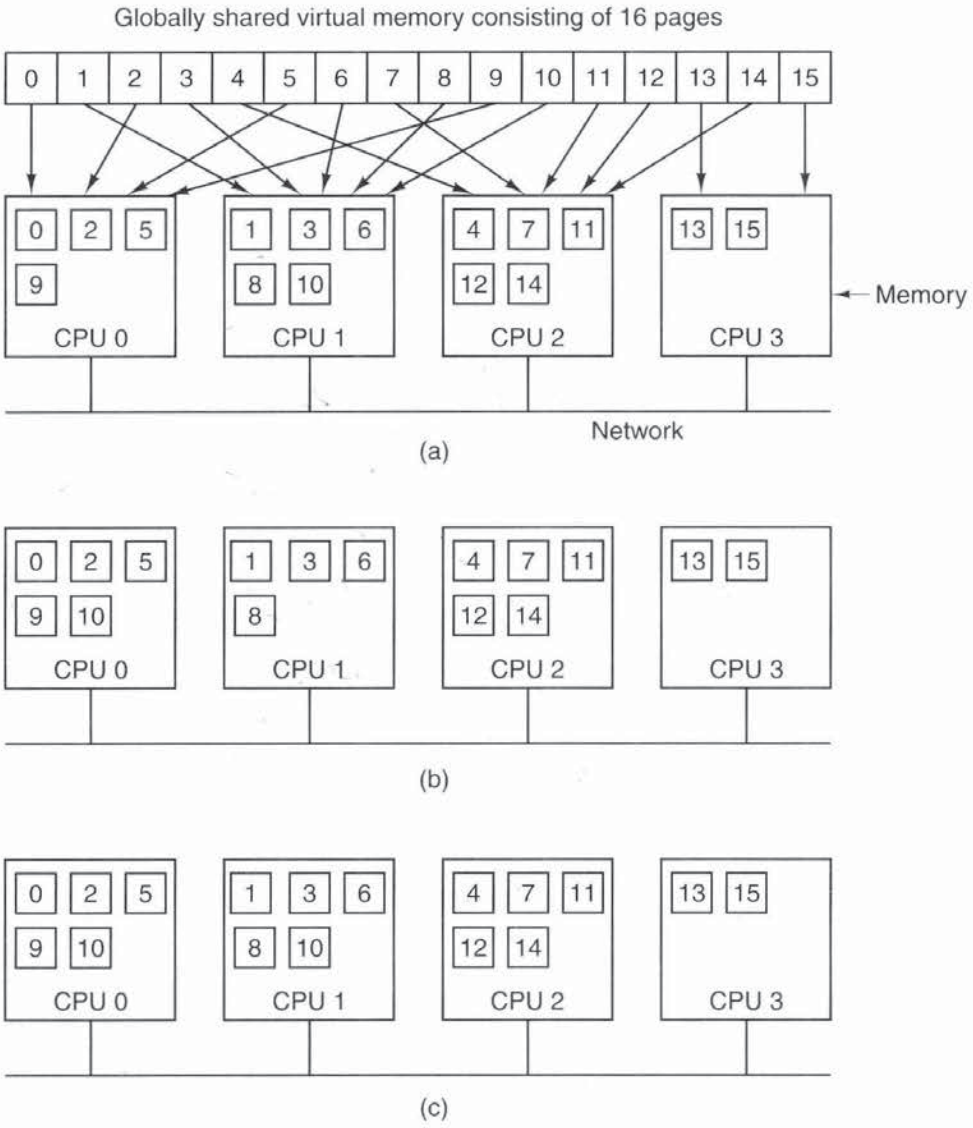


Figure 8-23. (a) Pages of the address space distributed among four machines. (b) Situation after CPU 1 references page 10. (c) Situation if page 10 is read only and replication is used.

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. 8-24. 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.

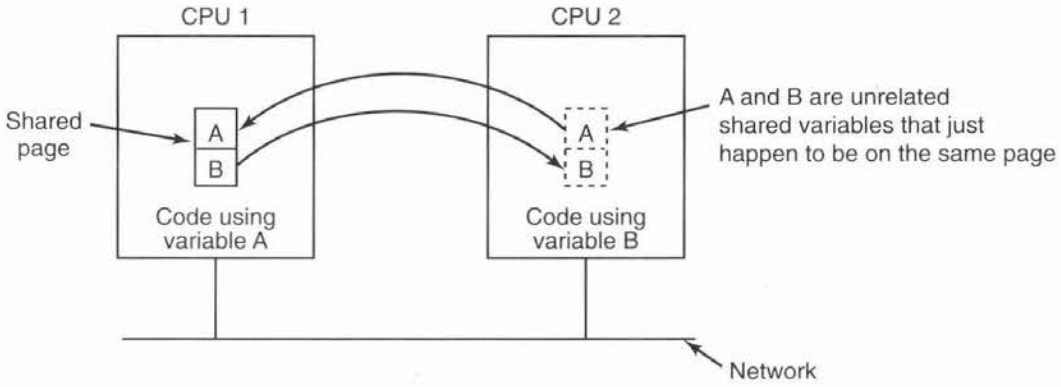


Figure 8-24. False sharing of a page containing two unrelated variables.

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 so it can 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.

Given that each node has its own processes, any local scheduling algorithm can be used. However, it is also possible to use gang scheduling, the same way it is used on a multiprocessor, 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 such as 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. 8-25 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).

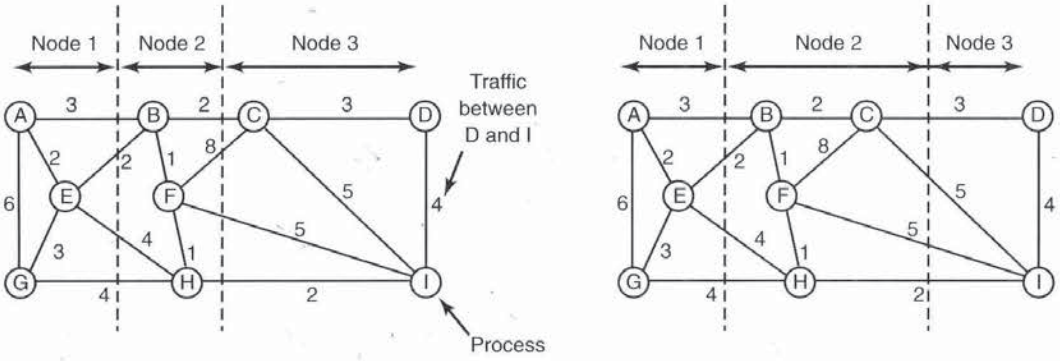


Figure 8-25. Two ways of allocating nine processes to three nodes.

In Fig. 8-25(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. 8-25(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).

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. 8-26(a).

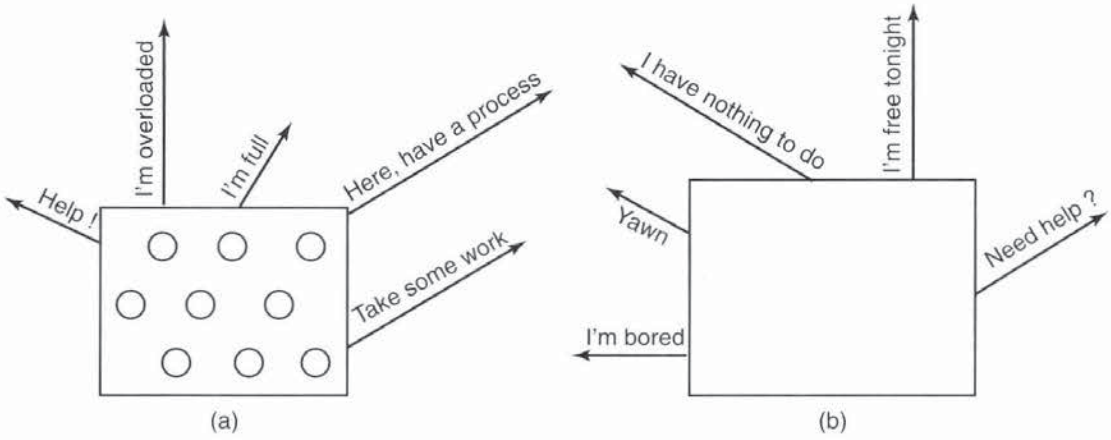


Figure 8-26. (a) An overloaded node looking for a lightly loaded node to hand off processes to. (b) An empty node looking for work to do.

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

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

A Receiver-Initiated Distributed Heuristic Algorithm

A complementary algorithm to the one given above, which is initiated by an overloaded sender, is one initiated by an underloaded receiver, as shown in Fig. 8-26(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.

A Bidding Algorithm

Another class of algorithms tries to turn the computer system into a miniature economy, with buyers and sellers of services and prices set by supply and demand (Ferguson et al., 1988). The key players in the economy are the processes, which must buy CPU time to get their work done, and nodes, which auction their cycles off to the highest bidder.

Each node advertises its approximate price by putting it in a publicly readable file. This price is not guaranteed, but gives an indication of what the service is worth (actually, it is the price that the last customer paid). Different nodes may have different prices, depending on their speed, memory size, presence of fast floating-point hardware, and other features. An indication of the service provided, such as expected response time, can also be published.

When a process wants to start up a child process, it goes around and checks out who is currently offering the service that it needs. It then determines the set of nodes whose services it can afford. From this set, it computes the best candidate, where "best" may mean cheapest, fastest, or best price/performance, depending on the application. It then generates a bid and sends the bid to its first choice. The bid may be higher or lower than the advertised price.

Processors collect all the bids sent to them and make a choice, presumably by picking the highest one. The winners and losers are informed, and the winning process is executed. The published price of the server is then updated to reflect the new going rate.

Although Ferguson et al. do not go into the details, such an economic model raises all kinds of interesting questions, among them the following. Where do processes get money to bid? Do they get regular salaries? Does everyone get the same monthly salary, or do deans get more than professors, who in turn get more than students? If new users are introduced into the system without a corresponding increase in resources, do prices get bid up (inflation)? Can nodes form cartels to gouge users? Are users' unions allowed? Is disk space also chargeable? How about printer output? Does printing pictures cost more than printing text due to the additional ink or toner used? The list goes on and on.

8.3 DISTRIBUTED SYSTEMS

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

To start with, the nodes of a multicomputer generally have a CPU, RAM, a network interface, and perhaps a hard disk for paging. In contrast, each node in a distributed system is a complete computer, with a full complement of peripherals. Next, the nodes of a multicomputer are normally in a single room so they can communicate by a dedicated high-speed network, whereas the nodes of a distributed system may be spread around the world. Finally, all the nodes of a multicomputer run the same operating system, share a single file system, and are under a common administration, whereas the nodes of a distributed systems may run different operating systems, each have their own file system, and be under different administrations. A typical example of a multicomputer is 512 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 8-27 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 8-27. 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 mind sets. 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* 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 Web site order a trip from New York to San Francisco, and only later discover 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 slightly different context is found in UNIX, where all I/O devices are made to look like files. Having keyboards, printers, and serial lines all be operated on the same way, with the same primitives, makes it easier to deal with them than having them all be conceptually different.

One way 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. 8-28. 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* 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, 2002). In the remainder of this chapter, we will first 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.

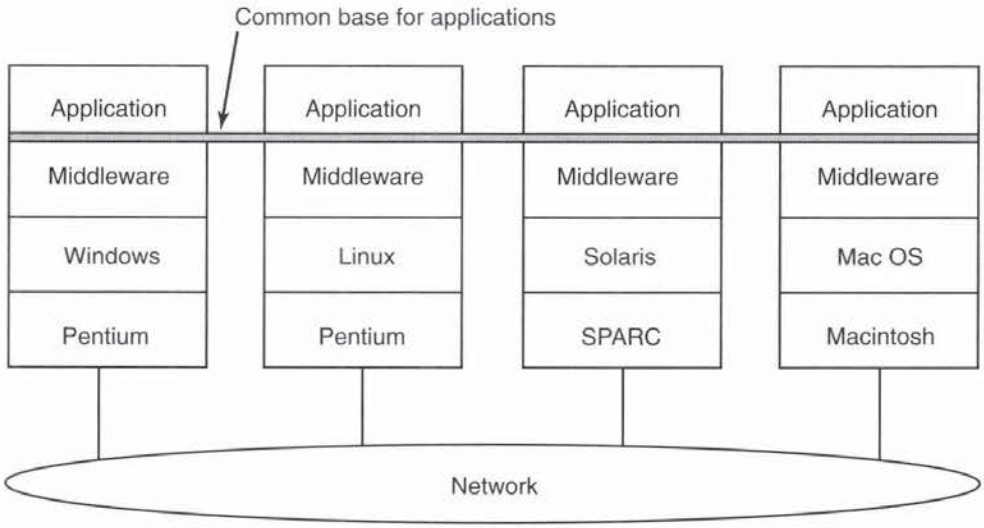


Figure 8-28. Positioning of middleware in a distributed system.

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 Networks**), that can be citywide, countrywide, or even 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 shown symbolically in Fig. 8-29(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.

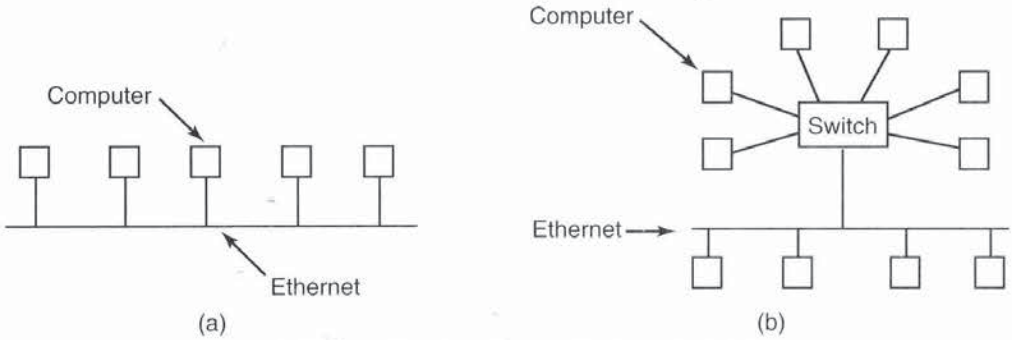


Figure 8-29. (a) Classic Ethernet. (b) Switched Ethernet.

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 0- to 1500-byte payload. 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 called **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 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. 8-29(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.

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 Dec. 1969 with three computers in California and one in

Utah. It was designed 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, laptops, palmtops, 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. 8-29(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 8-30 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.

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.

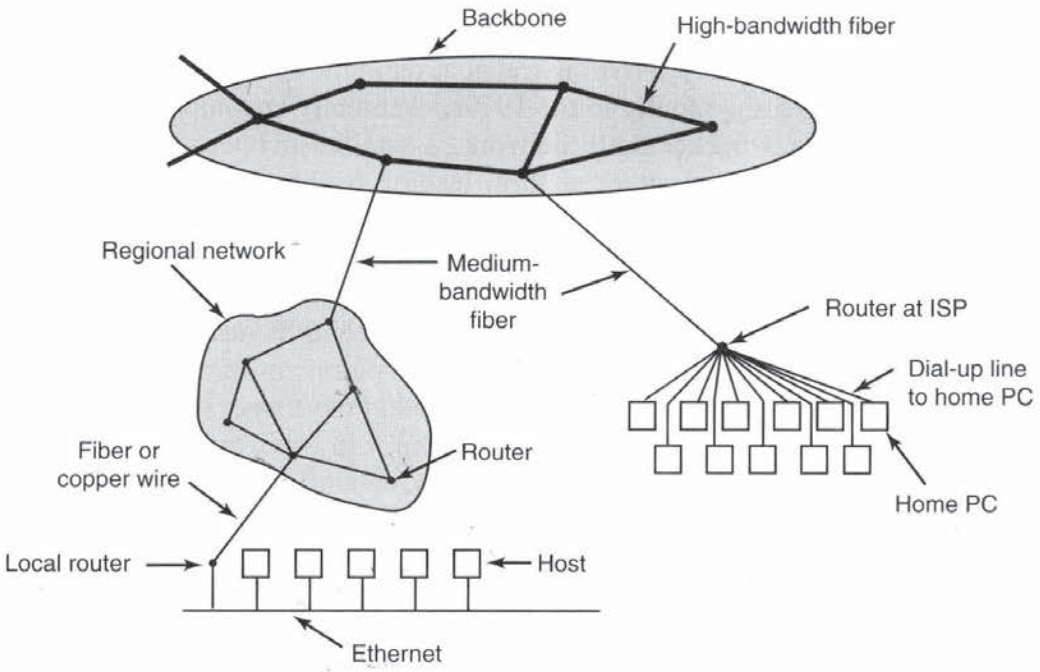


Figure 8-30. A portion of the Internet.

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 minor variations: 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, or 2048 1-byte messages. 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 timesharing system, a byte stream from the terminal to the computer is all that is needed.

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 8-31 summarizes the types of services discussed above.

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

	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 8-31. Six different types of network service.

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* (Tanenbaum, 1996).

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 routing in the Internet. 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 router, the router extracts the IP destination address and uses that for routing the packet.

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. 8-32.

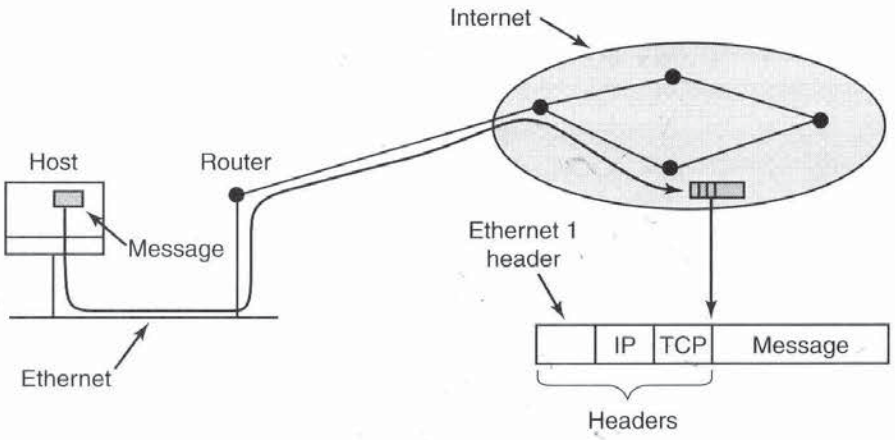


Figure 8-32. Accumulation of packet headers.

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 widely 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 has spread like wildfire all over the world since then.

The original paradigm behind the Web was very simple: every computer can hold one or more documents, called **Web pages**. Each Web page contains text, images, icons, sounds, movies, etc. 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 been grafted onto the Web recently, 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. 8-33.

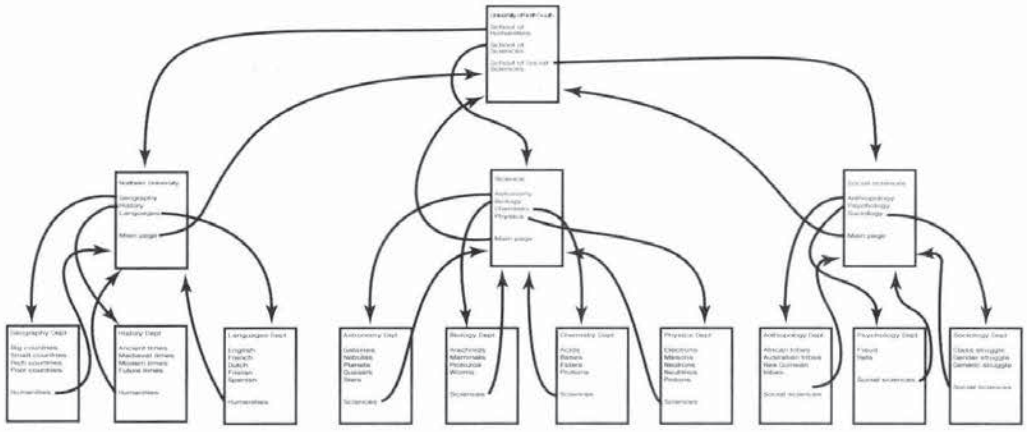


Figure 8-33. The Web is a big directed graph of documents.

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.

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 Web site

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 an example, suppose the URL provided is *http://www.acm.org/dl/faq.html* The browser then takes the following steps to get the page.

1. The browser asks DNS for the IP address of *www.acm.org*.
2. DNS replies with 199.222.69.151.
3. The browser makes a TCP connection to port 80 on 199.222.69.151.
4. It then sends a request asking for the file *dl/faq.html*.
5. The *www.acm.org* server sends the file *dl/faq.html*.
6. The TCP connection is released.
7. The browser displays all the text in *dl/faq.html*.
8. The browser fetches and displays all images in *dl/faq.html*.

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. 8-34(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. 8-34(b).

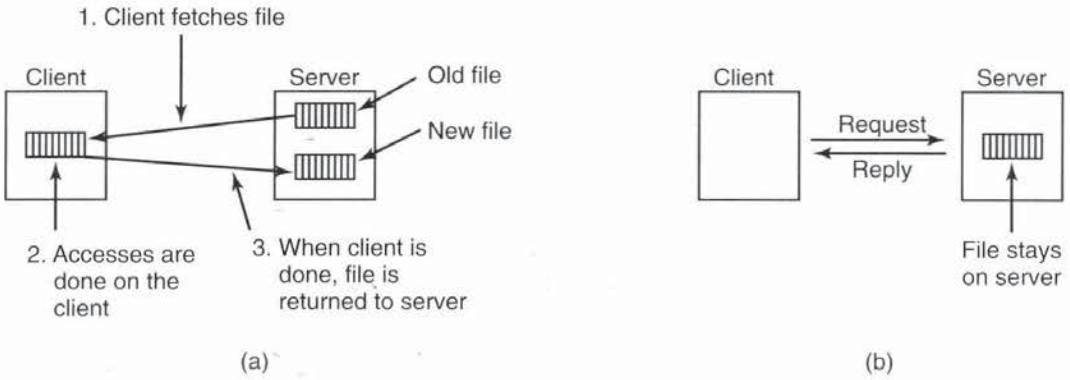


Figure 8-34. (a) The upload/download model. (b) The remote access model.

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 by this remark, consider Fig. 8-35. In Fig. 8-35(a) we show two file servers, each holding three directories and some files. In Fig. 8-35(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. 8-35(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. 8-35(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 [i.e., the model of Fig. 8-35(b)]. This property makes a system easier to program and understand.

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

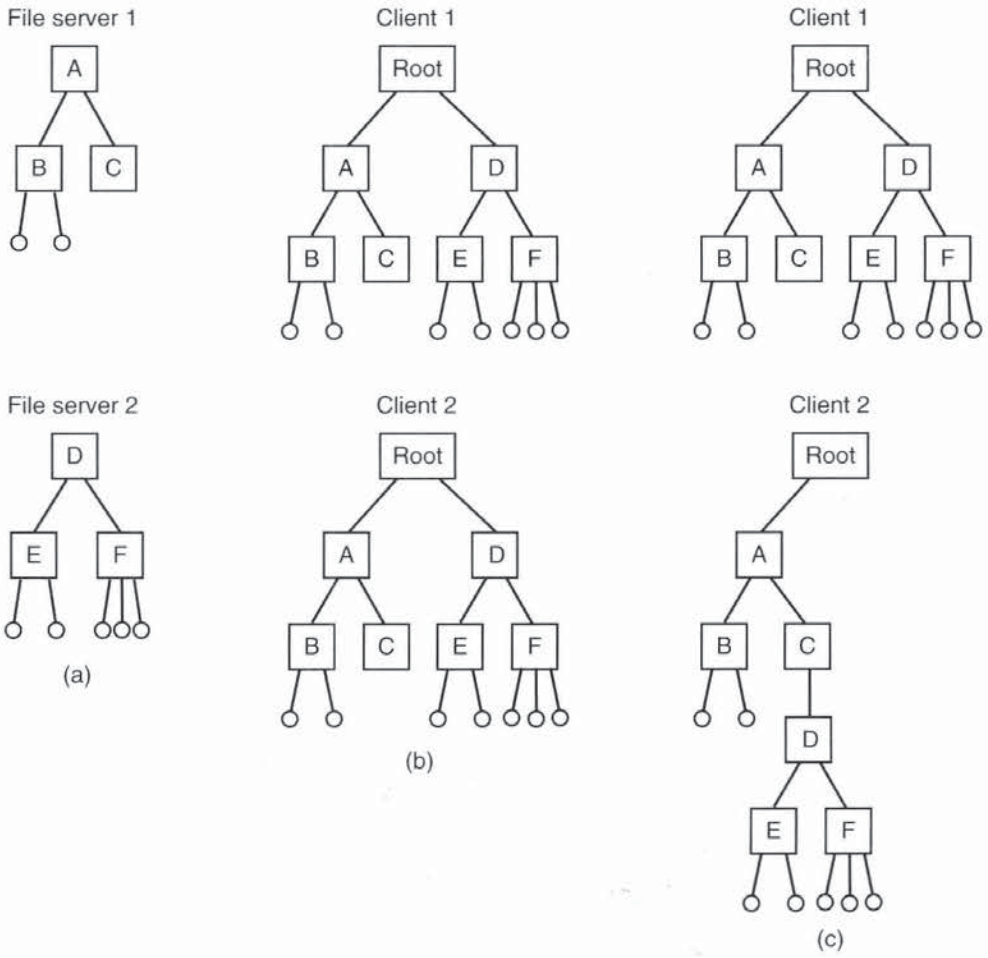


Figure 8-35. (a) Two file servers. The squares are directories and the circles are files. (b) A system in which all clients have the same view of the file system. (c) A system in which different clients may have different views of the file system.

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 have 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. 8-36(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 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. 8-36(b).

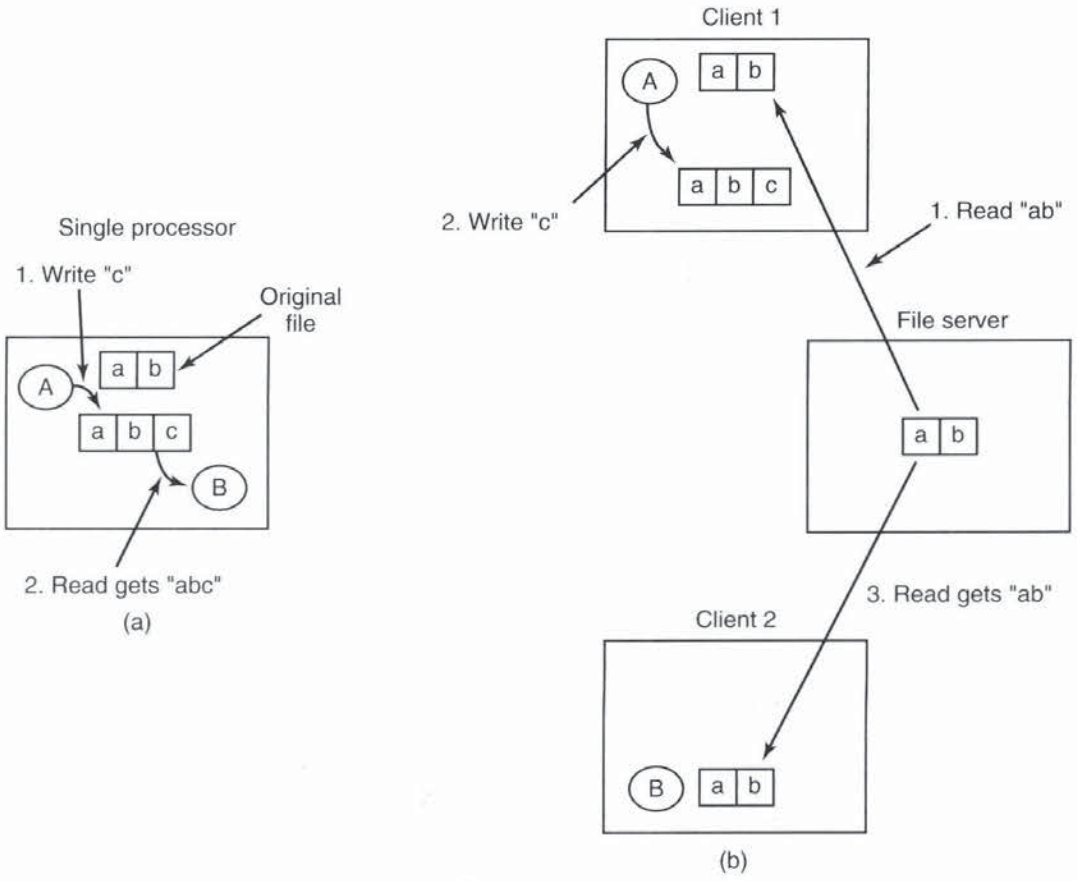


Figure 8-36. (a) Sequential consistency. (b) In a distributed system with caching, reading a file may return an obsolete value.

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. 8-36(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 of Fig. 8-34. 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.

AFS

Several file-system based middleware systems have been built and deployed. Below we will briefly discuss one (AFS) based on the upload/download model of Fig. 8-34(a). In Chap. 10, we will discuss one (NFS) based on the remote access model of Fig. 8-34(b).

AFS was designed and implemented at Carnegie-Mellon University (Howard et al. 1988; Morris et al., 1986; and Satyanarayanan et al., 1985). It was originally called the **Andrew File System** in honor of the university's first benefactors, Andrew Carnegie and Andrew Mellon. The goal of the project, which started in the early 1980s, was to provide every student and faculty member at CMU with a powerful personal workstation running UNIX, but with a shared file system. Here the file system was being used as middleware to turn a collection of workstations into a coherent system.

Each AFS user has a private workstation running a slightly modified version of UNIX. The modifications consist of adding a piece of code called **venus** to the kernel and running a file server called **vice** in user space (originally, venus also ran in user space, but was later moved into the kernel for performance reasons). The positions of *venus* and *vice* are shown in Fig. 8-37(a). User workstations are grouped into **cells** for administrative purposes. A cell might be a LAN or a collection of interconnected LANs or even an entire academic department.

The name space visible to user programs looks like a traditional UNIX tree, with the addition of a directories */cmu* and */cache*, as depicted in Fig. 8-37(b). The */cache* directory contains cached remote files. The */cmu* directory contains the names of the shared remote cells, below which are their respective file systems. In effect, remote file systems are mounted in */cmu*. The other directories and files are strictly local and are not shared. Symbolic links from local file names to shared files are permitted, as indicated by *sh* in Fig. 8-37(b).

The basic idea behind AFS is for each user to do as much as possible locally and interact as little as possible with the rest of the system. When a file is opened, the *venus* code traps the open call and downloads the entire file (or if it is a huge file, a large chunk of it) to the local disk and inserts it into the */cache* directory.

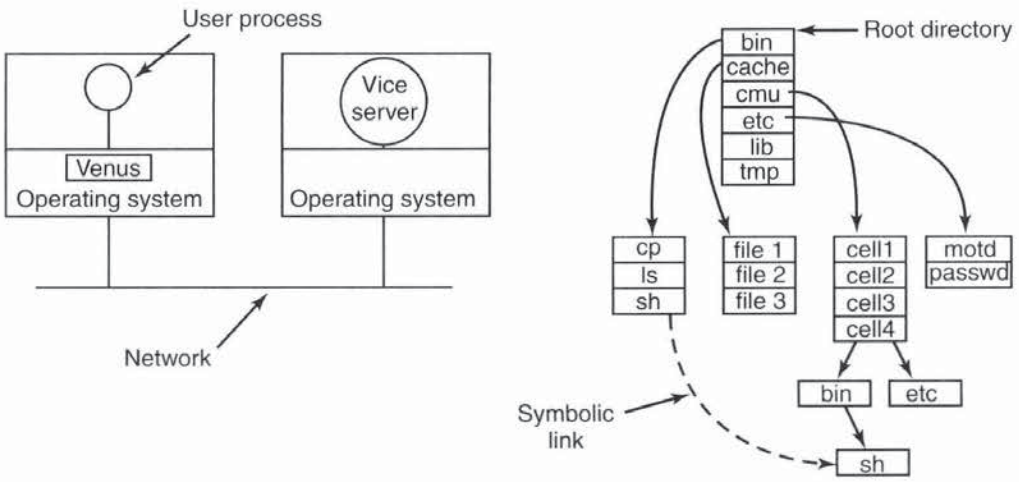


Figure 8-37. (a) The position of venus and vice in AFS. (b) A client's view of the file system.

The file descriptor returned by the open call refers to the file in */cache* so that subsequent read and write calls use the cached file.

The semantics offered by AFS are close to session semantics. When a file is opened, it is fetched from the appropriate server and placed in */cache* on the workstation's local disk. All reads and writes operate on the cached copy. When the file is closed, it is uploaded back to the server.

However, to prevent unsuspecting processes from using stale files in situations where it matters, when *venus* downloads a file into its cache, it tells *vice* whether or not it cares about subsequent opens by processes on other workstations. If it does, *vice* records the location of the cached file. If another process elsewhere in the system opens the file, *vice* sends a message to *venus* telling it to mark its cache entry as invalid and return the copy if it has been modified.

8.3.5 Shared 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.

CORBA

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, **ORB**s (**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 either come directly from the creating process, or more likely, by looking it up by name or by function in some kind of a 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 ref-

erences, and activating the object if it is invoked when it is not active. The arrangement of all these CORBA parts is shown in Fig. 8-38.

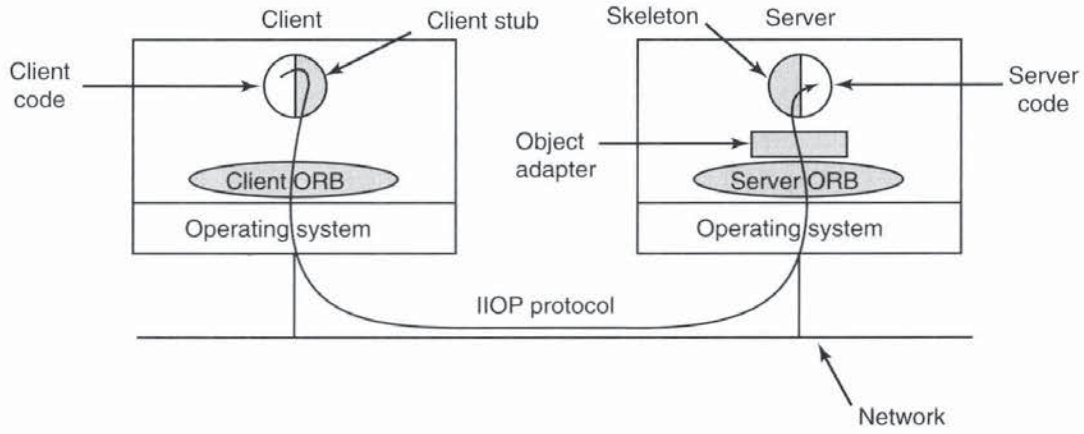


Figure 8-38. The main elements of a distributed system based on CORBA. The CORBA parts are shown in gray.

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 only functions acceptably in small-scale systems, such as to connect processes on one computer, one LAN or within a single company.

Globe

As an example of a distributed object system that was specifically designed to scale to a billion users and a trillion objects around the world, let us consider **Globe** (Van Steen et al., 1999a; Van Steen et al., 1999b). There are two key ideas to scaling to very large systems. The first is having replicated objects. If there is a single copy of a popular object that millions of users around the world want to access, the object will die under the weight of the requests. Think about an object that maintains stock prices or sports scores. Replicating this object allows the load to be spread over the replicas.

The second key idea is flexibility. In a worldwide system with a billion users, there is no way to get everyone to agree on one programming language, one replication strategy, one security model, or one anything else. The system has to allow different users and different objects to behave differently, while at the same time providing a coherent overall model. This is what Globe does.

Globe is also unusual in that, like DSM, it is based on the model of distributed shared memory, but now applied to a worldwide context. In principle, using normal page-based DSM on a worldwide system would work, only the performance

would be horrible, so Globe takes a different approach. Conceptually, the basic idea is that the world is full of objects, each one containing some (hidden) internal state plus methods for accessing the internal state in controlled ways. The secret to making shared memory scalable worldwide is prohibiting direct LOADs and STOREs to an object's internal state and forcing all accesses to go through the methods. Because a Globe object can actively be shared by many processes at the same time, it is also called a **distributed shared object**. The positioning of systems like Globe is shown in Fig. 8-22(c).

Now let us see how scalability and flexibility are implemented. Every Globe object has a class object that contains the actual code for its methods. Every object also has one (or more) interfaces, each of which contains (method pointer, state pointer) pairs. Thus given an object interface, which is a table full of pointers present in memory at run time, a process can invoke the object's n-th method by making a call to the procedure pointed to by the n-th pair in the interface table and passing it the corresponding state pointer as a parameter. The state pointer is needed so that if there are, say, two objects of class mailbox in memory, each one has its own interface, with shared method pointers but private state pointers, as shown in Fig. 8-39. In this example, the process has two open mailboxes, each of which shares the code for the four mailbox methods, but each of which has its own private state (the messages stored in the mailbox instance). One mailbox might be for business mail and the other for personal mail, for example.

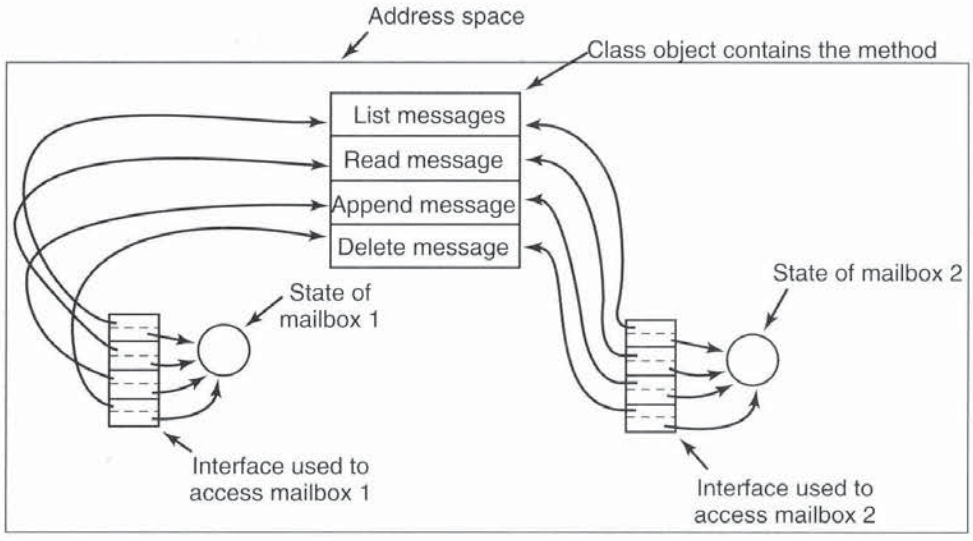


Figure 8-39. The structure of a Globe object.

The design of having interfaces be tables in memory at run time means that objects are not restricted to any particular language. This decision was made because a worldwide system will have many different people with many different

favorite languages. An object's methods may be written in C, C++, Java, or even assembly language if the object's owner so desires. The interfaces are there to shield the process from what is behind the method pointers. This mix-and-match design is more flexible than a single-language design present in some systems (e.g., only Java or only C++).

To use a Globe object, a process must first bind to it by looking it up and finding at least one contact address (e.g., IP address and port). A security check is made at binding time, and if the process is authorized to bind to the object, the object's class object (i.e., its code) is loaded into the caller's address space, a copy of its state is instantiated and a pointer to its (standard) interface is returned. Using the interface pointer, the process can now invoke methods on this instance of the object. Depending on the object, the state may be the default state or a copy of the current state taken from one of the other live copies.

Imagine the simplest possible object. It has one integer as state and two methods: *read* and *write* that operate on the integer. If multiple processes in different countries are simultaneously bound to the object, all of them have an interface table pointing to the class object containing the two methods (which was loaded at bind time), as illustrated in Fig. 8-40. Each process (potentially) also has a copy of the integer comprising the state. Any *read* method is just invoked locally, but *write*s are more complicated. If the object wants to maintain sequential consistency, it must provide a mechanism for doing so.

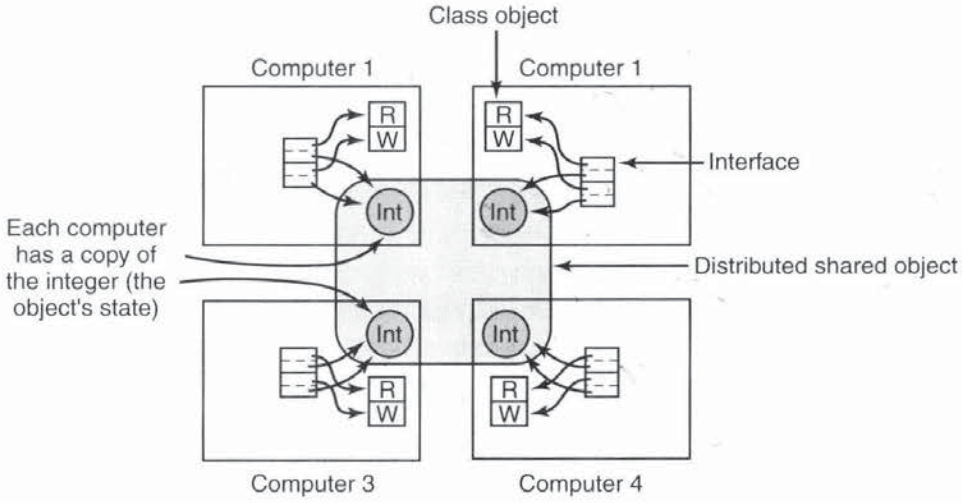


Figure 8-40. A distributed shared object can have its state copied on multiple computers at once.

One mechanism is to have a process called the **sequencer** whose job is to issue consecutive sequence numbers when requested to do so. To do a write, the *write* method might then first get a sequence number and then multicast a message

containing the sequence number, operation name, and parameter to all the other processes bound to the object. If two processes invoked *write* simultaneously, they would be assigned different consecutive sequence numbers. All processes must apply incoming methods in sequence number order, not in message arrival order. If a process gets sequence number 26 and the previous one was 24, it must wait for 25 before applying 26. If 25 does not show up within a certain time, the process must take action to locate and get it. This scheme guarantees that all writes are done in the same order on all replicas of the object, ensuring sequential consistency.

Using this technique works reasonably well, but not all objects need sequential consistency. Consider, for example, an object maintaining stock prices. If the market maker for stock 1 issues an updated price for it concurrently with another market maker issuing an update for stock 2, it is not essential that all copies of the object apply those two updates in the same order because they are independent. It is probably sufficient that all processes apply the stream of updates from each market maker in the order they were sent, but this goal can be achieved by including a sequence number generated by the sending process. No object-wide sequencer is needed here.

The above replication scheme, namely a replicated object with all copies being equal and any copy being allowed to issue updates after first getting a sequence number is only one of many replication protocols. Another one has one master copy of each object, plus some number of slave copies. All updates are sent to the object's master copy, which then applies the update and sends out the new state to all the slave copies.

A third object replication strategy is having only one copy holding the object's state, with all the other copies being stateless proxies. When a *read* or *write* is done at a proxy (e.g., a client machine), the request is forwarded to the copy holding the state and executed there.

The strength of Globe is that each object can have its own replication policy. Some objects can use active replication at the same time other objects are using master-slave replication or any other strategy an object needs. Also, each object can have its own policy concerning consistency, replica creation and removal, security, etc. This is possible because all the policies are handled inside the object. Users of the object are not even aware of it, and neither are the system administrators. This approach is in contrast to CORBA, which does not hide any of these policies inside objects, making it difficult to have 1000 different objects with 1000 different policies.

A Globe object can be implemented as shown in Fig. 8-41. This figure illustrates the subobjects from which a Globe object is composed. The control object accepts incoming method invocations and uses the other subobjects to get them done. The semantics subobject actually does the work required by the object's interface. It is the only part of the object's code that must be written by the object's programmer; all the rest can be taken from standard libraries, unless the

programmer wants a new strategy not currently available. The replication subobject's job is to manage replication. This module can be replaced to switch from active replication to master-slave replication or any other replication strategy without the rest of the object being affected. Similarly, the security subobject can be replaced to implement a new security policy (e.g., to switch from ACLs to capabilities) and the communication subobject can be replaced to change network protocols (e.g., from IP v4 to IP v6) without affecting the rest of the object.

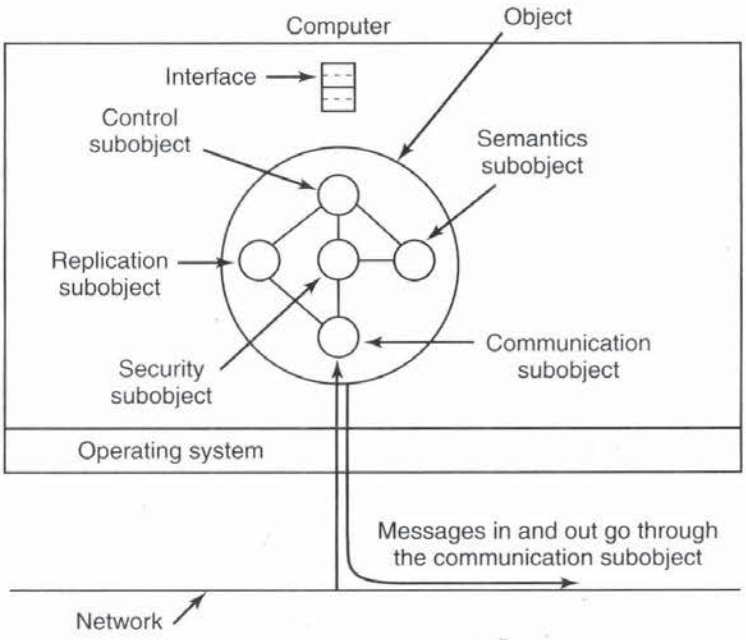


Figure 8-41. Structure of a Globe object.

To see how these subobjects interact, consider what happens when one of the object's methods is invoked. The code pointed to by the interface is in the control subobject, which then asks the replication subobject to do what it has to do. If the object is actively replicated, a sequence number is first acquired. Then the replication subobject tells all replicas (including its own) to actually do the work by invoking their semantics object. If the object is master-slave and the method invocation is on a slave, a message is sent to the master, and so on. At appropriate moments security checks are made by the security object (to see if the invocation is permitted, to see if outgoing data must be encrypted, etc.).

A key element of Globe is the **location service**, which allows objects to be looked up anywhere in the world. The location service is built as a tree, with object registrations being kept only in the node where the registration takes place. Pointers to this node are propagated up to the top of the tree so it is always possible to find the registration. Locality, partitioning of tree nodes, caching, and other techniques are used to make the scheme workable, even for mobile objects (Ballintijn et al., 2000; Van Steen et al., 1998a; and Van Steen et al., 1998b).

8.3.6 Coordination-Based Middleware

Our last paradigm for a distributed system is called **coordination-based middleware**. We will start with the Linda system, an academic research project that started the whole field, and then look at two commercial examples heavily inspired by it: publish/subscribe and Jini.

Linda

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 [and in Fig. 8-22(c)].

A **tuple** is like a structure in C or a record in Pascal. 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 8-42 shows three tuples as examples.

```
("abc", 2, 5)
("matrix-1", 1, 6, 3.14)
("family", "is-sister", "Stephany", "Roberta")
```

Figure 8-42. 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 caller 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 *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. 8-43. 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, only forwarding a tuple to a remote LAN if that remote LAN has at least one subscriber who wants the tuple. Doing this requires having the information routers exchange information about subscribers.

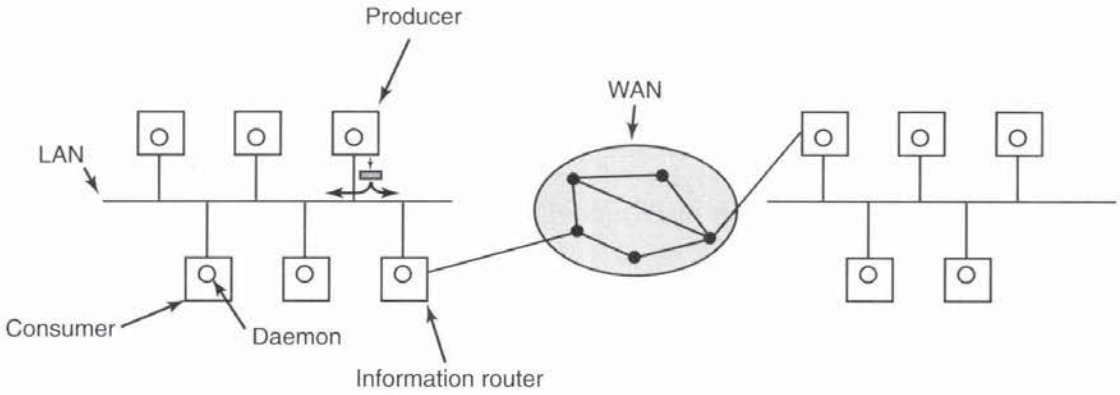


Figure 8-43. The publish/subscribe architecture.

Various kinds of semantics can be implemented, including reliable delivery and guaranteed delivery, even in the face 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."

Jini

For over 50 years, computing has been CPU-centric, with a computer being a freestanding device consisting of a CPU, some primary memory, and nearly always some mass storage such as a disk. Sun Microsystems' **Jini** (a variant spelling of genie) is an attempt to change that model to one that might be described as network-centric (Waldo, 1999).

The Jini world consists of a large number of self-contained Jini devices, each of which offers one or more services to the others. A Jini device can be plugged into a network and begin offering and using services instantly, with no complex installation procedure. Note that the devices are plugged into a *network*, not into a *computer* as is traditionally the case. A Jini device could be a traditional computer, but it could also be a printer, palmtop computer, cell phone, TV set, stereo, or other device with a CPU, some memory, and a (possibly wireless) network connection. A Jini system is a loose federation of Jini devices that may come and go at will, with no central administration.

When a Jini device wants to join the Jini federation, it broadcasts a packet on the local LAN or in the local wireless cell asking if there is a **lookup service** present. The protocol used to find a lookup service is the **discovery protocol** and is one of the few hardwired protocols in Jini. (Alternatively, the new Jini device can wait until one of the lookup service's periodic announcements comes by, but we will not treat this mechanism here.)

When the lookup service sees that a new device wants to register, it replies with a piece of code that can perform the registration. Since Jini is an all Java system, the code sent is in JVM (the Java Virtual Machine language), which all Jini devices must be capable of running, usually interpretively. The new device now runs the code, which contacts the lookup service and registers with it for some fixed period of time. Just before the time period expires, the device can reregister if it wishes. This mechanism means that a Jini device can just leave the system by shutting down and its previous existence will soon be forgotten, without the need for any central administration. The concept of registering for a fixed time interval is called acquiring a **lease**.

Note that since the code to register the device is downloaded into the device, it can be changed as the system evolves without affecting the hardware or software of the device. In fact, the device is not even aware of what the registration

protocol is. A part of the registration process that the device is aware of consists of it providing some attributes and proxy code that other devices will later use to access it.

A device or user looking for a particular service can ask the lookup service if it knows about one. The request may involve some of the attributes that devices use when registering. If the request is successful, the proxy that the device provided at registration time is sent back to the requester and is run to contact the device. Thus a device or user can talk to another device without knowing where it is or even what protocol it speaks.

Jini clients and services (hardware or software devices) communicate and synchronize using **JavaSpaces**, which are modeled on the Linda tuple space but with some important differences. Each JavaSpace consists of some number of strongly typed entries. Entries are like Linda tuples, except that they are strongly typed, whereas Linda tuples are untyped. Each entry consists of some number of fields, each of which has a basic Java type. For example, an entry of type employee might consist of a string (for the name), an integer (for the department), a second integer (for the telephone extension), and a Boolean (for works-full-time).

Just four methods are defined on a JavaSpace (although two of them have a variant form):

1. **Write**: put a new entry into the JavaSpace.
2. **Read**: copy an entry that matches a template out of the JavaSpace.
3. **Take**: copy and remove an entry that matches a template.
4. **Notify**: notify the caller when a matching entry is written.

The *write* method provides the entry and specifies its lease time, that is when it should be discarded. In contrast, Linda tuples stay until removed. A JavaSpace may contain the same entry multiple times, so it is not a mathematical set (just as in Linda).

The *read* and *take* methods provide a template for the entry being sought. Each field in the template can contain a specific value that must be matched, or can contain a "don't care" wildcard that matches all values of the appropriate type. If a match is found, it is returned, and in the case of *take*, it is also removed from the JavaSpace. Each of these JavaSpace methods has two variants, which differ in the case that no entry matches. One variant returns with a failure indication immediately; the other one waits until a timeout (given as a parameter) has expired.

The *notify* method registers interest in a particular template. If a matching entry is later entered, the caller's *notify* method is invoked.

Unlike Linda's tuple space, JavaSpace supports atomic transactions. Using them, multiple methods can be grouped together. They will either all execute or

none of them will execute. During the transaction, changes that are made to the JavaSpace are not visible outside the transaction. Only when the transaction commits, do they become visible to other callers.

JavaSpace can be used for synchronization between communicating processes. For example, in a producer-consumer situation, the producer puts items in a JavaSpace as it produces them. The consumer removes them with *take*, blocking if none are available. JavaSpace guarantees that each of the methods is executed atomically, so there is no danger of one process trying to read an entry that has only been half entered.

8.4 RESEARCH ON MULTIPLE PROCESSOR SYSTEMS

In this chapter we have looked at three kinds of multiple processor systems: multiprocessors, multicomputers, and distributed systems. Let us also look briefly at the research in these three areas. Most of the research on multiprocessors relates to the hardware, in particular, how to build the shared memory and keep it coherent. However, there has also been some research on using virtual machine monitors on multiprocessors (Bugnion et al., 1997) and on resource management on multiprocessors (Govil et al., 1999). Thread scheduling is also an issue in terms of the scheduling algorithm (Arora et al., 1998; and Philbin et al., 1996), and also in terms of contention for the run queue (Dandamudi, 1997).

Multicomputers are much easier to build than multiprocessors. All that is needed is a collection of PCs or workstations and a high-speed network. For this reason, they are a popular research topic at universities. A lot of the work relates to distributed shared memory in one form or another, sometimes page-based but sometimes entirely in software (Carter et al., 1995; Feeley et al., 1995; Johnson et al., 1995; Itzkovitz and Schuster, 1999; Scales and Gharachorloo, 1997; and Stets et al., 1997). Optimizing user-level communication is also a research topic (Von Eicken et al., 1995). So is load balancing (Harchol-Balter and Downey, 1996).

There are also many papers on distributed systems, for example on middleware (Bernstein, 1996), objects (Dogac et al., 1998), wireless systems (Liu et al., 1996), mobile agents (Chen et al., 2000), programming environments (Jo, 1999), distributed multimedia (Mourlas, 2000), theory (Buchs and Guelfi, 2000), and Web caching (Wolman et al., 1999), among others. Distributed file systems (Alexandrov et al., 1998; Hartman and Ousterhout, 1995; and Thekkath et al., 1997) and mobile file systems (Segarra and Andri, 1999) are also also popular.

8.5 SUMMARY

Computer systems can be made faster and more reliable by using multiple CPUs. Three organizations for multiCPU systems are multiprocessors, multicomputers, 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. The 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 timesharing, 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 call, 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 of middleware include document-based, file-based, object-based, and coordination-based middleware. Some examples are the World Wide Web, AFS, CORBA, Globe, Linda, and Jini.

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 radiotelescope 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 3B in the omega network of Fig. 8-5 breaks. Who is cut off from whom?

5. How is signal handling done in the model of Fig. 8-7?

6. 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?

7. Rewrite the enter_region code of Fig. 2-22 using the pure read to reduce thrashing induced by the TSL instruction.

8. 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?

9. 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?

10. In the text, it was suggested that a binary exponential backoff algorithm be used between 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?

11. 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?

12. 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.

13. Fig. 8-12 was said to depict a timesharing environment. Why is only one process (*A*) shown in part (b)?

14. Affinity scheduling reduces cache misses. Does it also reduce TLB misses? What about page faults?

15. 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.

16. 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.

17. 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?

18. 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 for a 64-byte packet being sent from source to destination be 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?

19. 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.

20. How must the implementation of send and receive differ between a shared memory multiprocessor system and a multicomputer, and how does this affect performance?

21. 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 byte/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?

22. When a procedure is scooped up from one machine and placed on another to 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?

23. In a DSM system, when a page fault occurs, the needed page has to be located. List two possible ways to find the page.

24. Consider the processor allocation of Fig. 8-25. Suppose that process *H* is moved from node 2 to node 3. What is the total weight of the external traffic now?

25. 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 nontrivial problems that have to be solved to make this work.

26. Why is there a limit to cable length on an Ethernet network?

27. In Fig. 8-28, 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?

28. Fig. 8-31 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.

29. DNS names have a hierarchical structure, such as *cs.uni.edu* or *sales.general-widget.com*. 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. Make a proposal how the DNS database could be maintained in practice.

30. In the discussion of how URLs are processed by a browser, it was stated that connections are made to port 80. Why?

31. Can the URLs used in the Web exhibit location transparency? Explain your answer.

32. 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.

33. 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.

34. In AFS, whole files are cached on the client machines. Suppose that there is only so much disk space allocated for cached files and the allocation is full. When a new file is requested, what should be done? Give an algorithm for doing it.

35. When multiple processes need access to data, in what way is object-based access better than shared memory?

36. 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.

37. 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.

38. 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 variables your client stub will accept.

39. Write two programs to simulate load balancing on a multicomputer. The first program should set up m processes distributed across n machines according to an initialization file. Each process should have running time chosen at random from a Gaussian

distribuion whose mean and standard deviation are parameters of the simulation. At the end of each run, the process creates some number of new processes, chosen from a Poisson distribution. When a process exits, the CPU must decide whether to give away processes or try to find new processes. The first program should use the sender-initiated algorithm to give away work if it has more than k processes total on its machine. The second program should use the receiver-initiated algorithm to fetch work when needed. Make any other reasonable assumptions you need, but state them clearly.

9

SECURITY

Many companies possess valuable information that they guard closely. 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), legal (e.g., documents about a potential merger or takeover), among many other possibilities. Frequently this information is protected by having a uniformed guard at the building entrance who checks to see that all people entering the building are wearing a proper badge. In addition, many offices may be locked and some file cabinets may be locked as well to ensure that only authorized people have access to the information.

As more and more of this information is stored in computer systems, the need to protect it is becoming increasingly important. Protecting this 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 as being a normal and acceptable phenomenon. In the following sections we will look at a variety of issues concerned with security and protection, some of which have analogies to real-world protection of information on paper, but some of which are unique to computer systems. In this chapter we will examine computer security as it applies to operating systems.

9.1 THE SECURITY ENVIRONMENT

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 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.

Security has many facets. Three of the more important ones are the nature of the threats, the nature of intruders, and accidental data loss. We will now look at these in turn.

9.1.1 Threats

From a security perspective, computer systems have three general goals, with corresponding threats to them, as listed in Fig. 9-1. The first one, **data confidentiality**, is concerned with having secret data remain secret. More specifically, if the owner of some data has decided that these data are only to be made available to certain people and no others, the system should guarantee that release of the data to unauthorized people does not occur. As a bare minimum, the owner should be able to specify who can see what, and the system should enforce these specifications.

Goal	Threat
Data confidentiality	Exposure of data
Data integrity	Tampering with data
System availability	Denial of service

Figure 9-1. Security goals and threats.

The second goal, **data 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 as well. If a system cannot guarantee that data deposited in it remain unchanged until the owner decides to change them, it is not worth much as an information system.

The third goal, **system 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-services attacks is much harder.

Another aspect of the security problem 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? 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 Intruders

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 **intruders** or sometimes **adversaries**. Intruders act in two different ways. Passive intruders just want to read files they are not authorized to read. Active intruders are more malicious; they want to make unauthorized changes to data. When designing a system to be secure against intruders, it is important to keep in mind the kind of intruder one is trying to protect against. Some common categories are

1. Casual prying by nontechnical users. Many people have personal computers on their desks that are connected to a shared file server, and human nature being what it is, some of them will read other people's electronic mail and other files if no barriers are placed in the way. Most UNIX systems, for example, have the default that all newly created files are publicly readable.
2. Snooping by insiders. Students, system programmers, operators, and other technical personnel often consider it to be a personal challenge to break the security of the local computer system. They often are highly skilled and are willing to devote a substantial amount of time to the effort.
3. Determined attempts to make money. Some bank programmers have attempted to steal from the bank they were working for. Schemes have varied from changing the software to truncate rather than round interest, keeping the fraction of a cent for themselves, to siphoning

off accounts not used in years, to blackmail ("Pay me or I will destroy all the bank's records.").

4. Commercial or military espionage. Espionage refers to a serious and well-funded attempt by a competitor or a foreign country to steal programs, trade secrets, patentable ideas, technology, circuit designs, business plans, and so forth. Often this attempt will involve wiretapping or even erecting antennas directed at the computer to pick up its electromagnetic radiation.

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 security and protection clearly depends on who the enemy is thought to be.

Another category of security pest that has manifested itself in recent years is the virus, which will be discussed at length below. Basically a virus is a piece of code that replicates itself and (usually) does some damage. In a sense, the writer of a virus is also an intruder, often with high technical skills. The difference between a conventional intruder and a virus is that the former refers to a person who is personally trying to break into a system to cause damage whereas the latter is a program written by such a person and then released into the world hoping it causes damage. Intruders try to break into specific systems (e.g., one belonging to some bank or the Pentagon) to steal or destroy particular data, whereas a virus usually causes more general damage. In a sense, an intruder is like someone with a gun who tries to kill a specific person; a virus writer is more like a terrorist bomber who just wants to kill people in general, rather than some particular person.

9.1.3 Accidental Data Loss

In addition to threats caused by malicious intruders, valuable data can be lost by accident. Some of the common causes of accidental data loss are

1. Acts of God: fires, floods, earthquakes, wars, riots, or rats gnawing tapes or floppy disks.
2. Hardware or software errors: CPU malfunctions, unreadable disks or tapes, telecommunication errors, program bugs.
3. Human errors: incorrect data entry, wrong tape or disk mounted, wrong program run, lost disk or tape, or some other mistake.

Most of these can be dealt with by maintaining adequate backups, preferably far away from the original data. While protecting data against accidental loss may seem mundane compared to protecting against clever intruders, in practice, probably more damage is caused by the former than the latter.

9.2 BASICS OF CRYPTOGRAPHY

A little knowledge of cryptography may be useful for understanding parts of this chapter and some subsequent ones. 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 some of these (e.g., Kaufman et al., 1995; and Pfleeger, 1997). Below we 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 the **ciphertext** in such a way that only authorized people know how to convert it back to the 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 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, this category 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.

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. 9-2.

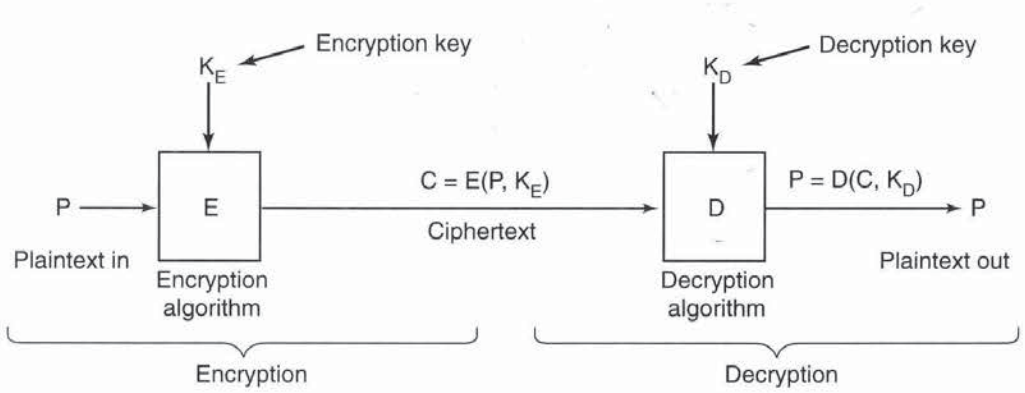


Figure 9-2. Relationship between the plaintext and the ciphertext.

9.2.1 Secret-Key Cryptography

To make this clearer, consider an encryption algorithm in which each letter is replaced by a different letter, for example, all *A*s are replaced by *Q*s, all *B*s are replaced by *W*s, all *C*s are replaced by *E*s, 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 above, the 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 $26! \simeq 4 \times 10^{26}$ 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*, etc. 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, and vice versa. Such systems are called **secret-key cryptography** or **symmetric-key cryptography**. Although monoalphabetic substitution ciphers are worthless, other symmetric key algorithms are known and are relatively secure if the keys are long enough. For serious security, probably 1024-bit keys should be used, giving a search space of $2^{1024} \simeq 2 \times 10^{308}$ keys. Shorter keys may thwart amateurs, but not major governments.

9.2.2 Public-Key Cryptography

Secret key systems are efficient because the amount of computation required to encrypt or decrypt a message is manageable, but 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 × 314159265358979?
Question 2: What is the square root of 3912571506419387090594828508241?

Most sixth graders 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 big numbers is much easier for a computer to do than factoring 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.

9.2.3 One-Way Functions

There are various situations that we will see later in which 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.

9.2.4 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 email messages 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 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 are **MD5** (**Message Digest**), which produces a 16-byte result (Rivest, 1992) and **SHA** (**Secure Hash Algorithm**), which produces a 20-byte result (NIST, 1995).

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. 9-3. 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.

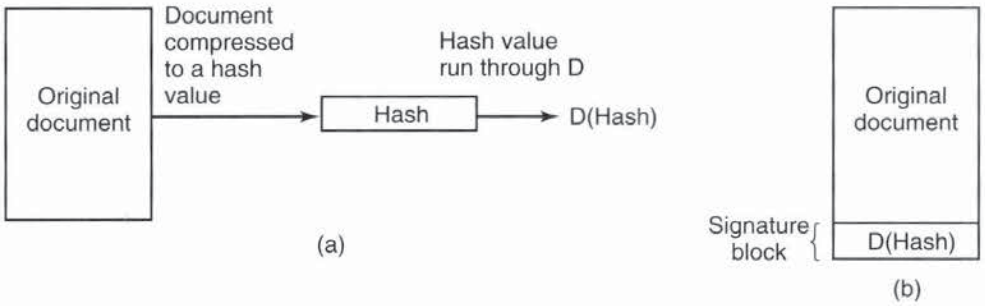


Figure 9-3. (a) Computing a signature block. (b) What the receiver gets.

When the document and hash arrive, the receiver first computes the hash of the document using MD5 or SHA, as agreed upon in advance. The receiver than 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 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.

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.

9.3 USER AUTHENTICATION

Now that we have some cryptographic background, let us start looking at security issues in operating systems. When a user logs into a computer, the operating system normally wishes to determine who the user is, a process called **user authentication**.

User authentication is one of those things we meant by "ontogeny recapitulates phylogeny" in Sec. 1.2.5. 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 Windows 2000, again require a secure login. Using a personal computer to access servers on a LAN (local area network) or one's account at an e-commerce Web site always requires logging in. Thus the subject of secure login has gone through several cycles, and is once again an important subject.

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.

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.

People who want to cause trouble on a particular system have to first log in to that system, which means getting past whichever authentication procedure is used. In the popular press, these people are called **hackers**. 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 **crackers**.

9.3.1 Authentication Using Passwords

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 terminal. With Windows 2000, 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 2000 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. 9-4. In Fig. 9-4(a), a successful login is shown, with system output in upper case and user input in lower case. In Fig. 9-4(b), a failed attempt by a cracker to log into System *A* is shown. In Fig. 9-4(c) a failed attempt by a cracker to log into System *B* is shown.

LOGIN: ken	LOGIN: carol	LOGIN: carol
PASSWORD: FooBar	INVALID LOGIN NAME	PASSWORD: Idunno
SUCCESSFUL LOGIN	LOGIN:	INVALID LOGIN
		LOGIN:
(a)	(b)	(c)

Figure 9-4. (a) A successful login. (b) Login rejected after name is entered. (c) Login rejected after name and password are typed.

In Fig. 9-4(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. 9-4(c), the cracker is always asked for a password and gets no feedback about whether the login name itself is valid. All he learns is that the login name plus password combination tried is wrong.

How Crackers Break In

Most crackers break in by just calling up the target computer 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 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 U.S. 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 might 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, and short strings of random characters. 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. A similar result was obtained by Klein (1990).

Lest anyone think that better quality users pick better quality passwords, rest assured that they do not. A 1997 survey of passwords used in the financial district of London revealed that 82% could be guessed easily. Commonly used passwords were sexual terms, abusive expressions, people's names (often a family member or a sports star), vacation destinations, and common objects found around the office (Kabay, 1997). Thus a cracker can compile a list of potential login names and a list of potential passwords without much work.

Does it really matter if passwords are easy to guess? Yes. In 1998, the *San Jose Mercury News* reported that a Berkeley resident, Peter Shipley, had set up several unused computers as **war dialers**, which dial 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. He estimated that a determined cracker could break into about 75% of the other ones (Denning, 1999).

The combination of a war dialer and password guessing can be deadly. An Australian cracker wrote a program that systematically dialed all the numbers at a telephone exchange and then attempted to break in using password guessing, notifying him when it succeeded. Among the many systems he broke into was a Citibank computer in Saudi Arabia, which allowed him to obtain credit card numbers and credit limits (in one case, $5 million) and transaction records (including at least one visit to a brothel). A cracker colleague of his also broke into the bank and collected 4000 credit card numbers (Denning, 1999). If such information were misused, the bank would undoubtedly emphatically and vigorously deny that it could possibly be at fault, claiming that the customer must have disclosed the information.

An alternative to using a war dialer is to attack computers over the Internet. Every computer on the Internet has a 32-bit IP address used to identify it. People usually write these addresses in **dotted decimal notation** as *w.x.y.z*, where each of the four components of the IP address is an integer from 0 to 255 in decimal. A cracker can easily test if some computer has this IP address and is up and running by typing

```
ping w.x.y.z
```

If the computer is alive, it will respond and the *ping* program will tell how long the roundtrip time was in milliseconds (although some sites now disable *ping* to prevent this kind of attack). It is easy to write a program to ping large numbers of IP addresses systematically, analogous to what a war dialer does. If a live computer is found at *w.x.y.z*, the cracker can attempt to break in by typing

```
telnet w.x.y.z
```

If the connection attempt is accepted (which it may not be, since not all system administrators welcome random logins over the Internet), the cracker can start trying login names and passwords from his lists. At first, it is trial and error. However, the cracker may eventually be able to break in a few times and capture the password file (located in */etc/passwd* on UNIX systems and often publicly readable). Then he will begin to collect statistical information about login name usage frequencies to optimize future searches.

Many telnet daemons break the underlying TCP connection after some number of unsuccessful login attempts in order to slow down crackers. Crackers

respond to that by starting up many threads in parallel, working on different target machines at once. Their goal is to make as many tries per second as the outgoing bandwidth will allow. From their point of view, having to spray them over many machines being attacked simultaneously is not a serious disadvantage.

Instead of pinging machines in IP-address order, a cracker may wish to target a specific company, university, or other organization, say, the University of Foobar at *foobar.edu*. To find out what IP addresses they use, all he has to do is type

```
dnsquery foobar.edu
```

and he will get a list of some of their IP addresses. (Alternatively, the programs *nslookup* or *dig* can also be used.) Since many organizations have 65,536 consecutive IP addresses (a common allocation unit in the past), once he knows the first 2 bytes of their IP addresses (which *dnsquery* supplies), it is straightforward to ping all 65,536 of them to see which ones respond and which ones accept telnet connections. From there on, it is back to guessing login names and passwords, a subject we have already discussed.

Needless to say, the entire process of starting with a domain name, finding the first 2 bytes of its IP addresses, pinging all of them to see which ones are alive, checking to see if any accept telnet connections, and then trying statistically likely (login name, password) pairs is a process that lends itself very well to automation. It will take many, many tries to break in, but if there is one thing that computers are very good at, it is repeating the same sequence of commands over and over until the cows come home. A cracker with a high-speed cable or DSL connection can program the break in process to run all day long and just check back once in a while to see what has showed up.

A telnet attack is clearly better than a war dialer attack since it goes much faster (no dialing time) and is much cheaper (no long distance telephone charges), but it only works for machines that are on the Internet and accept telnet connections. Nevertheless, many companies (and nearly all universities) do accept telnet connections so employees on a business trip or at a different branch office (or students at home) can log in remotely.

Not only are user passwords often weak, but sometimes the root password is too. In particular, some installations never bother to change the default passwords that systems are shipped with. Cliff Stoll, an astronomer at Berkeley, had observed irregularities on his system, and laid a trap for the cracker who had been trying to get in (Stoll, 1989). He observed the session shown in Fig. 9-5 typed by a cracker who had already broken into one machine at the Lawrence Berkeley Laboratory (LBL) and was trying to get into another one. The uucp (UNIX to UNIX Copy Program) account is used for intermachine network traffic and has superuser power, so the cracker was now in a U.S. Dept. of Energy machine as superuser. Fortunately, LBL does not design nuclear weapons although its sister lab at Livermore, does. One hopes their security is better, but there is little reason

to believe that since another nuclear weapons lab, Los Alamos, lost a hard disk full of classified information in 2000.

```
LBL> telnet elxsi
ELXSI AT LBL
LOGIN: root
PASSWORD: root
INCORRECT PASSWORD, TRY AGAIN
LOGIN: guest
PASSWORD: guest
INCORRECT PASSWORD, TRY AGAIN
LOGIN: uucp
PASSWORD: uucp
WELCOME TO THE ELXSI COMPUTER AT LBL
```

Figure 9-5. How a cracker broke into a U.S. Dept. of Energy computer at LBL.

Once a cracker has broken into a system and become superuser, it may be possible to install a **packet sniffer**, software that examines all incoming and outgoing network packets looking for certain patterns. An especially interesting pattern to look for is people on the compromised machine logging into remote machines, especially as superuser there. This information can be squirreled away in a file for the cracker to pick up at his leisure later. In this way, a cracker who breaks into one machine with weak security can often leverage this into a way to break into other machines with stronger security.

Increasingly many break ins are being done by technically naive users who are just running scripts they found on the Internet. These scripts either use brute force attacks of the type described above, or try to exploit known bugs in specific programs. Real hackers refer to them as **script kiddies**.

Usually, the script kiddie has no particular target and no particular information he is trying to steal. He is just looking for machines that are easy to break into. Some of the scripts even pick a network to attack by chance, using a random network number (in the upper partof the IP address). They then probe all the machines on the network to see which ones respond. Once a database of valid IP addresses has been acquired, each machine is attacked in sequence. As a consequence of this methodology, it can happen that a brand new machine at a secure military installation can be attacked within hours of its being attached to the Internet, even though no one but the administrator even knows about it yet.

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, 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.

However, this scheme can also 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.

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 stored in the password file, as shown in Fig. 9-6 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(Dog4238)* 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.

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*,

Bobbie, 4238, e(Dog4238)
Tony, 2918, e(6%%TaeFF2918)
Laura, 6902, e(Shakespeare6902)
Mark, 1694, e(XaB@Bwcz1694)
Deborah, 1092, e(LordByron,1092)

Figure 9-6. The use of salt to defeat precomputation of encrypted passwords.

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, some modern versions of UNIX make the password file itself unreadable but provide a program to look up entries upon request, adding just enough delay to greatly slow down any attacker. The combination of salting the password file and making it unreadable except indirectly (and slowly) can generally withstand most attacks on it.

Improving Password Security

Although salting the password file protects against crackers who try to precompute a large list of encrypted passwords and thus break many passwords at once, it does little to protect a user *David* whose password is also *David*. A cracker can still just try guessing passwords one at a time. Educating users about the need for strong passwords is critical, but few installations do it. One step further than user education is to have the computer help. Some computers have a program that generates random easy-to-pronounce nonsense words, such as *fotally*, *garbungy*, or *bipitty* that can be used as passwords (preferably with some upper case and special characters thrown in). The program that users call to install or change their password can also give a warning when a poor password is chosen. Among other items it might complain about are

1. Passwords should be a minimum of seven characters.
2. Passwords should contain both upper and lower case letters.
3. Passwords should contain at least one digit or special character.
4. Passwords should not be dictionary words, people's names, etc.

A lenient password program might just carp; a strict one could reject the password and demand a better one. The password program could also make a suggestion, as discussed above.

Some operating systems require users to change their passwords regularly, to limit the damage done if a password leaks out. The trade-off here is that if users have to change their password too often, they are going to run out of good ones

and start picking easy ones. If prevented from picking easy ones, they will forget them and start writing them down on sticky notes attached to their monitors, which becomes a major security hole itself.

One-Time Passwords

The most extreme form of changing the passwords all the time is the **one-time password**. 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 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 computational infeasible to find x. The input and output should be the same length, for example, 128 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 are

1. Who is Marjolein's sister?
2. On what street was your elementary school?
3. What did Mrs. Woroboff 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 terminal has real computing power, such as a personal computer, a personal digital assistant, or a cellular telephone, 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.

9.3.2 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 terminal or computer. Normally, the user must not only insert the card, but 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 sent to the central computer. Often the information contains the user's password (e.g., PIN code) so the terminal can do 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 each, 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 an integrated circuit (chip) on them. These cards can be further 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 EEPROM 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 about $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 EEPROM, 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 $5 to $50, 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, pay phones, 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 potential 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 (called a **protocol**) with a central computer to authenticate the user. For example, a user wishing to buy things at an e-commerce Web site 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 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 EEPROM 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. 9-7. 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.

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

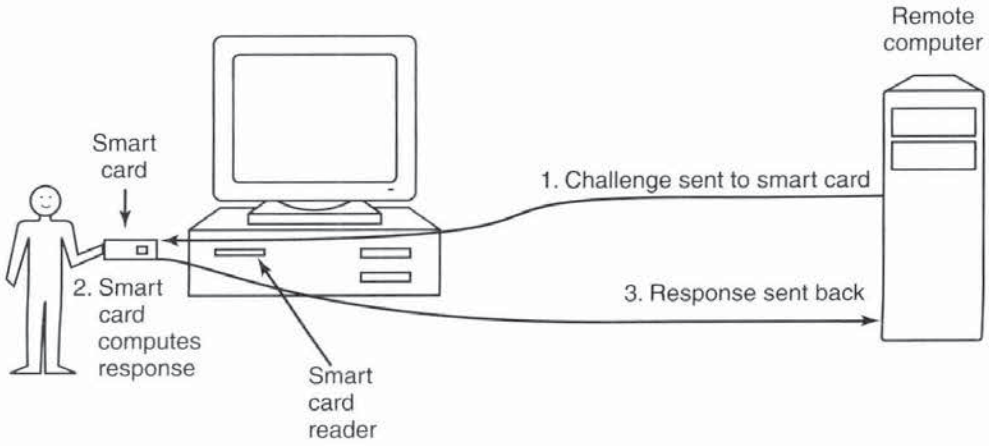


Figure 9-7. Use of a smart card for authentication.

protocol is broken, a new one can be installed worldwide instantly. 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 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.

9.3.3 Authentication Using Biometrics

The third authentication method measures physical characteristics of the user that are hard to forge. These are called **biometrics** (Pankanti et al., 2000). For example, a fingerprint or a voiceprint reader in the terminal 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 not exact, so it is difficult to index them and then search the index. Also, two people may have the same characteristics, so requiring the measured characteristics to match those of a specific user is stronger than just requiring it 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 much over time. For example, a person's voice may be different due to a cold and a face may look different due to a beard or make-up 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 now. Finger length analysis is surprisingly practical. When this is used, each terminal has a device like the one of Fig. 9-8. The user inserts his hand into it, and the length of all his fingers is measured and checked against the database.

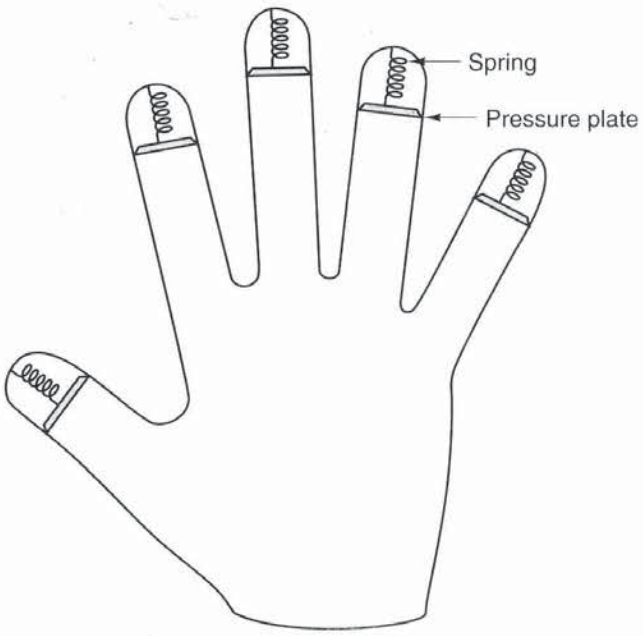


Figure 9-8. A device for measuring finger length.

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 gaining in popularity is retinal pattern analysis. Every person has a different pattern of retinal blood vessels, even identical twins.

They can be accurately photographed by a camera 1 meter from the subject, without the person even being aware of it. The amount of information in a retinal scan is much more than in a fingerprint, and it can be coded in about 256 bytes,

Any technique that relies on images is subject to spoofing. For example, a person could approach the ATM machine camera wearing dark glasses to which photographs of someone else's retinas were attached. After all, if the ATM's camera can take a good retinal photo at 1 meter, other people can do it too, and at greater distances using telephoto lenses. For this reason, video cameras are generally used instead of still cameras, and the pulsations normally present in the retinal blood vessels are looked for.

A somewhat different technique is signature analysis. The user signs his name with a special pen connected to the terminal, 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 (Markowitz, 2000). 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 tape 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.

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 this way. Suppose that someone comes up with a tiny device capable of doing an instant urinalysis, thereby providing a foolproof identification. Each terminal 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.

The same could be said of 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. The 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.

9.3.4 Countermeasures

Computer installations that are really serious about security, something that frequently happens the day after an intruder has broken in and done major damage, often take steps to make unauthorized entry much harder. For example, a company could have a policy that people working in the patent department are only allowed to log in from 8 A.M. to 5 P.M. Monday through Friday and then only from a machine in the patent department connected to the company LAN. Any attempt by a patent department employee to log in at the wrong time or from the wrong place would be treated as an attempted break in.

Dial-up telephone lines can also be made secure as follows. Anyone can dial up and log in, but after a successful login, the system immediately breaks the connection and calls the user back at an agreed upon number. This measure means than an intruder cannot just try breaking in from any phone line; only the user's (home) phone will do. In any event, with or without call back, the system should take at least 5 seconds to check any password typed in on a dial-up line, and should increase this time after several consecutive unsuccessful login attempts, in order to reduce the rate at which intruders can try. After three failed login attempts, the line should be disconnected for 10 minutes and security personnel notified.

All logins should be recorded. When a user logs in, the system should report the time and terminal of the previous login, so he can detect possible break ins.

The next step up is laying baited traps to catch intruders. A simple scheme is to have one special login name with an easy password (e.g., login name: guest, password: guest). Whenever anyone logs in using this name, the system security specialists are immediately notified. All commands the intruder types are immediately displayed on the security manager's screen so he can see exactly what the intruder is up to.

Other traps can be easy-to-find bugs in the operating system and similar things, designed for the purpose of catching intruders in the act. Stoll (1989) has written an entertaining account of the traps he set to track down a spy who broke into a university computer in search of military secrets.

9.4 ATTACKS FROM INSIDE THE SYSTEM

Once a cracker has logged into a computer, he can start doing damage. If the computer has good security, it may only be possible to harm the user whose account has been broken, but often this initial entry can be leveraged to break into more accounts later. In the following sections, we will look at some attacks that can be set up by someone already logged in, either a cracker who has gotten in illictly or possibly a legitimate user with a grudge against someone.

9.4.1 Trojan Horses

One hoary insider attack is the **Trojan horse**, in which a seemingly innocent program contains code to perform an unexpected and undesirable function. This function might be modifying, deleting or encrypting the user's files, copying them to a place where the cracker can retrieve them later, or even sending them to the cracker or a temporary safe hiding place via email or FTP. To have the Trojan horse run, the person planting it first has to get the program carrying it executed. One way is to place the program on the Internet as a free, exciting new game, MP3 viewer, "special" porno viewer, or something else likely to attract attention, and encourage people to download it. When it runs, the Trojan horse procedure is called and can do anything the user can do (e.g., delete files, open network connections, etc.). Note that this ploy does not require the author of the Trojan horse to break into the victim's computer.

There are other ways to trick the victim into executing the Trojan horse program as well. 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 takes a look to see if there is a program named */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 so a cracker could 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.

A 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 /usr/mal
ls –l
```

to see what Mal has in his home directory. Since some shells try the local directory before working through *$PATH*, the superuser may have just invoked Mal's Trojan horse with superuser power. The Trojan horse could make */usr/mal/bin/sh* SETUID root. All it takes is two system calls: chown to change the owner of */usr/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, such as *Quicken*, 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.

In the second scam, the Trojan horse first turns off the modem's sound, then dials a 900 (pay) number, again, preferably in a far-away country, such as Moldova (part of the former Soviet Union). If the user was online when the Trojan horse was started, then the 900 phone number in Moldova needs to be a (very expensive) Internet provider, so the user will not notice and perhaps stay online for hours. Neither of these techniques is hypothetical; both have happened and are reported in (Denning, 1999). In the latter one, 800,000 minutes of connect time to Moldova were run up before the U.S. Federal Trade Commission managed to get the plug pulled and filed suit against three people on Long Island. They eventually agreed to return $2.74 million to 38,000 victims.

9.4.2 Login Spoofing

Somewhat related to Trojan horses is **login spoofing**. It works as follows. Normally, when no one is logged in on a UNIX terminal or workstation on a LAN, a screen such as Fig. 9-9(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 is started.

Now consider this scenario. Mal writes a program to display the screen of Fig. 9-9(b). It looks amazingly like the screen of Fig. 9-9(a), except that this is not the system login program running, but a phony one written by Mal. Mal now



Figure 9-9. (a) Correct login screen. (b) Phony login screen.

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. 9-9(a). The user assumes that she made a typing error and just logs in again. This time it works. But in the meantime, Mal has acquired another (login name, password) pair. By logging in at many terminals and starting the login spoofer on all of them, he can collect many passwords.

The only real way to guard against this is to have the login sequence start with a key combination that user programs cannot catch. Windows 2000 uses CTRL-ALT-DEL for this purpose. If a user sits down at a terminal and starts out by 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.

9.4.3 Logic Bombs

Another insider attack in these times of high employee mobility is the **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 operating system. As long as the programmer feeds it its daily password, it 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 this blackmail and to 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).

9.4.4 Trap Doors

Another security hole caused by an insider is the **trap 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. 9-10(a). The trap door would be the change to Fig. 9-10(b). 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 trap 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 trap door simply bypasses the whole authentication process.

```
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 9-10. (a) Normal code. (b) Code with a trap door inserted.

One way for companies to prevent trap doors 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 trap 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.

9.4.5 Buffer Overflow

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 programming language (because programmers like it and it can be compiled extremely efficiently).

Unfortunately, no C compiler does array bounds checking. Consequently, the following code sequence, while not legal, is also not checked:

```
int i;
char c[1024];
i = 12000;
c[i] = 0;
```

The result is that some byte of memory 10,976 bytes outside the array *c* is overwritten, possibly with disastrous consequences. No check is performed at run time to prevent this error.

This property of C leads to attacks of the following kind. In Fig. 9-11(a), we see the main program running, with its local variables on the stack. At some point it calls a procedure *A*, as shown in Fig. 9-11(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 to allocate storage for its local variables.

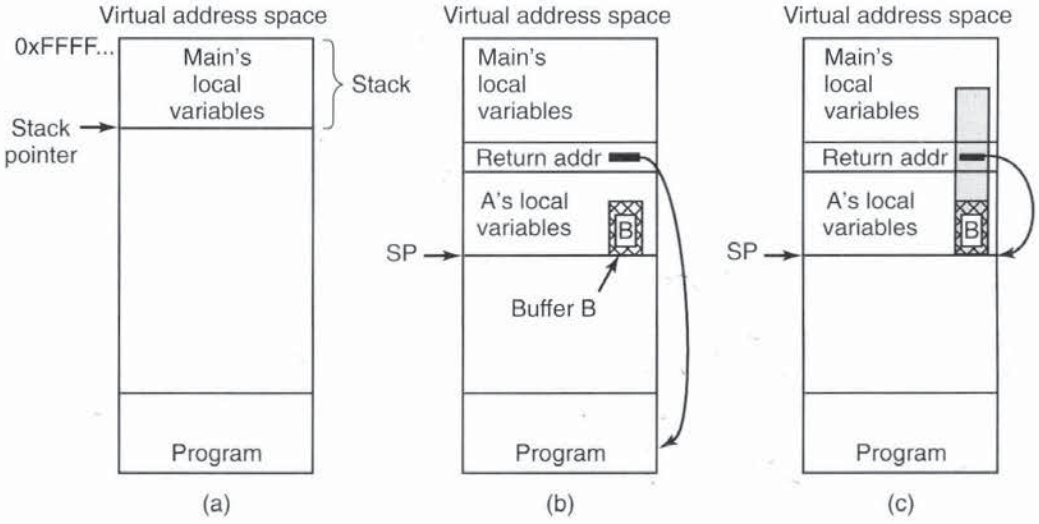


Figure 9-11. (a) Situation when the main program is running. (b) After the procedure *A* has been called. (c) Buffer overflow shown in gray.

Suppose that the job of *A* requires acquiring the full file path (possibly by concatenating the current directory path with a file name) and then opening it or doing something else with it. *A* has a fixed-size buffer (i.e., array) *B* to hold a file name, as shown in Fig. 9-11(b). Using a fixed-size buffer to hold the file name is much easier to program than first determining the actual size and then dynamically allocating enough storage. If the buffer is 1024 bytes, that should handle all file names, right? Especially if the operating system limits file names (or better yet, full paths) to a maximum of no more than 255 characters.

Unfortunately, this reasoning contains a fatal flaw. Suppose that the user of the program provides a file name that is 2000 characters long. When the file name is used, it will fail to open, but the attacker does not care. When the procedure copies the file name into the buffer, the name overflows the buffer and overwrites memory as shown in the gray area of Fig. 9-11(c). Worse yet, if the file name is long enough, it also overwrites the return address, so when *A* returns, the return address is taken from the middle of the file name. If this address is random junk, the program will jump to a random address and probably crash within a few instructions.

But what if the file name does not contain random junk? What if it contains a valid binary program and the layout has been very, very carefully made so that the word overlaying the return address just happens to be the address of the start of the program, for example, the address of *B*? What will happen is that when *A* returns, the program now in *B* will start executing. In effect, the attacker has inserted code into the program and gotten it executed.

This same trick works with things other than file names. It works with very long environment strings, user input, or anything else where the programmer has created a fixed-size buffer to handle a user-supplied string that was expected to be short. By providing a long handcrafted string containing a program, it may be possible to get the program onto the stack and then get it executed. 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.

Now comes the really bad part. Suppose that the program being attacked is SETUID root in UNIX (or has administrator power in Windows 2000, which is effectively the same thing). The inserted code can now make a couple of system calls to convert the attacker's shell file on the disk into SETUID root, so that when it is executed it has superuser power. Alternatively, it can now map in a specially prepared shared library that can do all kinds of damage. Or it can simply issue an exec system call to overlay the current program with the shell, creating a shell with superuser powers. 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.

Detecting that a program has buffer overflow problems is easy: just feed it 10,000-character file names, 100-digit salaries, or something equally unexpected to see if it dumps core. The next step is to analyze the core dump to see where the long stream is stored. From there, figuring out which character overwrites the return address is not so difficult. If the source code is available, as it is for most UNIX programs, the attack is even easier because the layout of the stack is known in advance. The attack can be defended against by fixing the code to explicitly check the length of all user-supplied strings before stuffing them into fixed-length buffers. Unfortunately, the fact that some program is vulnerable to this kind of attack generally shows up after a successful attack.

9.4.6 Generic Security Attacks

The usual way to test a system's security is to hire a group of experts, known as **tiger teams** or **penetration teams**, to see if they can break in. Hebbard et al. (1980) tried the same thing with graduate students. In the course of the years, these penetration teams have discovered a number of areas in which systems are likely to be weak. Below we have listed some of the more common attacks that are often successful. While these were originally designed to attack timesharing systems, they can often also be used to attack LAN servers and other shared machines. When designing a system, be sure it can withstand attacks like these.

1. Request memory pages, disk space, or tapes and just read them. Many systems do not erase them before allocating them, and they may be full of interesting information written by the previous owner.
2. Try illegal system calls, or legal system calls with illegal parameters, or even legal system calls with legal but unreasonable parameters such as file names thousands of characters long. Many systems can easily be confused.
3. Start logging in and then hit DEL, RUBOUT or BREAK halfway through the login sequence. In some systems, the password checking program will be killed and the login considered successful.
4. Try modifying complex operating system structures kept in user space (if any). In some systems (especially on mainframes), to open a file, the program builds a large data structure containing the file name and many other parameters and passes it to the system. As the file is read and written, the system sometimes updates the structure itself. Changing these fields can wreak havoc with the security.
5. Look for manuals that say "Do not do *X*." Try as many variations of *X* as possible.
6. Convince a system programmer to add a trap door by skipping certain vital security checks for any user with your login name.
7. All else failing, the penetrator might find the system administrator's secretary and pose as a poor user who has forgotten his password and needs it quickly. An alternative approach is an out-and-out bribe of the secretary. The secretary probably has easy access to all kinds of wonderful information, and is usually poorly paid. Do not underestimate problems caused by personnel.

These and other attacks are discussed by Linde (1975). While this is an old paper, the attacks described in it often still work.

9.4.7 Famous Security Flaws

Just as the transportation industry has the *Titanic*, the *Hindenburg*, and the *Concorde* disasters, operating system designers also have a few things they would rather forget about. In this section we will look at some interesting security problems that have occurred in three different operating systems: UNIX, TENEX, and OS/360.

Famous Security Flaws in UNIX

The UNIX utility *lpr*, which prints a file on the line printer, has an option to remove the file after it has been printed. In early versions of UNIX it was possible for anyone to use *lpr* to print and then have the system remove the password file.

Another way to break into UNIX was to link a file called *core* in the working directory to the password file. The intruder then forced a core dump of a SETUID program, which the system wrote on the *core* file, that is, on top of the password file. In this way, a user could replace the password file with one containing a few strings of his own choosing (e.g., command arguments).

Yet another subtle flaw in UNIX involved the command

```
mkdir foo
```

Mkdir, which was a SETUID program owned by the root, first created the i-node for the directory *foo* with the system call mknod and then changed the owner of *foo* from its effective UID (i.e., root) to its real UID (the user's UID). When the system was slow, it was sometimes possible for the user to quickly remove the directory i-node and make a link to the password file under the name *foo* after the mknod but before the chown. When *mkdir* did the chown, it made the user the owner of the password file. By putting the necessary commands in a shell script, they could be tried over and over until the trick worked.

Famous Security Flaws in TENEX

The TENEX operating system used to be very popular on the DEC-10 computers. It is no longer used, but it will live on forever in the annals of computer security due to the following design error. TENEX supported paging. To allow users to monitor the behavior of their programs, it was possible to instruct the system to call a user function on each page fault.

TENEX also used passwords to protect files. To access a file, a program had to present the proper password to the operating system at the time the file was opened. The operating system checked passwords one character at a time, stopping as soon as it saw that the password was wrong. To break into TENEX an intruder would carefully position a password as shown in Fig. 9-12(a), with the first character at the end of one page, and the rest at the start of the next page.

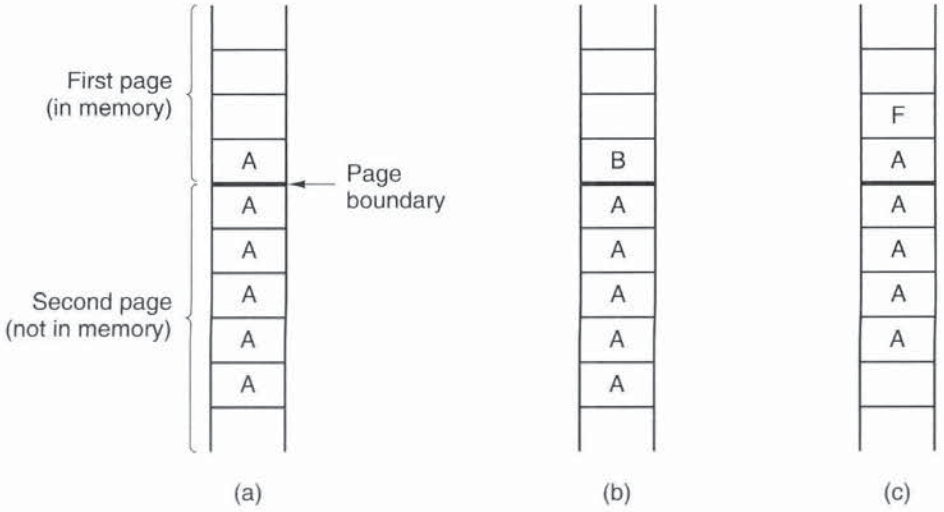


Figure 9-12. The TENEX password problem.

The next step was to make sure that the second page was not in memory, for example, by referencing so many other pages that the second page would surely be evicted to make room for them. Now the program tried to open the victim's file, using the carefully aligned password. If the first character of the real password was anything but *A*, the system would stop checking at the first character and report back with ILLEGAL PASSWORD. If, however, the real password did begin with *A*, the system continued reading and got a page fault, about which the intruder was informed.

If the password did not begin with *A*, the intruder changed the password to that of Fig. 9-12(b) and repeated the whole process to see if it began with *B*. It took at most 128 tries to go through the whole ASCII character set and thus determine the first character.

Suppose that the first character was an *F*. The memory layout of Fig. 9-12(c) allowed the intruder to test strings of the form *FA*, *FB*, and so on. Using this approach it took at most $128n$ tries to guess an *n-character* ASCII password, instead of 128^n.

Famous Security Flaws in OS/360

Our last flaw concerns OS/360. The description that follows is slightly simplified but preserves the essence of the flaw. In this system it was possible to start up a tape read and then continue computing while the tape drive was transferring data to the user space. The trick here was to carefully start up a tape read and then do a system call that required a user data structure, for example, a file to read and its password.

The operating system first verified that the password was indeed the correct one for the given file. After that it went back and read the file name again for the

actual access (it could have saved the name internally, but it did not). Unfortunately, just before the system went to fetch the file name the second time, the file name was overwritten by the tape drive. The system then read the new file, for which no password had been presented. Getting the timing right took some practice, but it was not that hard.

9.4.8 Design Principles for Security

By now it should be clear that designing a secure operating system is not a trivial matter. People have been working on this problem for decades without much success. As far back as 1975, researchers have identified some general principles that should be used as a guide to designing secure systems (Saltzer and Schroeder, 1975). A brief summary of their ideas (based on experience with MULTICS) is given below. These ideas are as valid now as when they were first stated.

First, the system design should be public. Assuming that the intruders do not know how the system works serves only to delude the designers. The intruders will find out sooner or later, and if the protection is compromised by this knowledge, the system is sunk.

Second, the default should be no access. Errors in which legitimate access is refused will be reported much faster than errors in which unauthorized access is allowed. When in doubt, say "No."

Third, check for current authority. The system should not check for permission, determine that access is permitted, and then squirrel away this information for subsequent use. Many systems check for permission when a file is opened, and not afterward. This means that a user who opens a file, and keeps it open for weeks, will continue to have access, even if the owner has long since changed the file protection or maybe even tried to delete the file.

Fourth, give each process the least privilege possible. If an editor has only the authority to access the file to be edited (specified when the editor is invoked), editors with Trojan horses will not be able to do much damage. This principle implies a fine-grained protection scheme. We will discuss such schemes later in this chapter.

Fifth, the protection mechanism should be simple, uniform, and built into the lowest layers of the system. Trying to retrofit security to an existing insecure system is nearly impossible. Security, like correctness, is not an add-on feature.

Sixth, the scheme chosen must be psychologically acceptable. If users feel that protecting their files is too much work, they just will not do it. Nevertheless, they will complain loudly if something goes wrong. Replies of the form "It is your own fault" will generally not be well received.

To this list, we would like to add one other principle that has been gained by decades of hard-won experience:

Keep the design simple

If the system is elegant and simple, was designed by a single architect, and has a few guiding principles that determine the rest, it has a chance of being secure. If the design is a mess, with no coherence and many fundamental concessions to ancient insecure systems in the name of backward compatibility, it is going to be a security nightmare. You can design a system with many features (options, user-friendliness, etc.) but a system with many features is a big system. And a big system is potentially an insecure system. The more code there is, the more security holes and bugs there will be. From a security perspective, the simplest design is the best design.

9.5 ATTACKS FROM OUTSIDE THE SYSTEM

The threats discussed in the previous sections were largely caused from the inside, that is, perpetrated by users already logged in. However, for machines connected to the Internet or another network, there is a growing external threat. A networked computer can be attacked from a distant computer over the network. In nearly all cases, such an attack consists of some code being transmitted over the network to the target machine and executed there doing damage. As more and more computers join the Internet, the potential for damage keeps growing. In the following sections we will look at some of the operating systems aspects of these external threats, primarily focusing on viruses, worms, mobile code, and Java applets.

It is hard to open a newspaper these days without reading about another computer virus or worm attacking the world's computers. They are clearly a major security problem for individuals and companies alike. In the following sections we will examine how they work and what can be done about them.

I was somewhat hesitant to write this section in so much detail, lest it give some people bad ideas, but existing books give far more detail and even include real code (e.g., Ludwig, 1998). 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.

Unlike, say, game programmers, successful virus writers tend not to seek publicity after their products have made their debut. Based on the scanty evidence there is, it appears that most are high school or college students or recent graduates who wrote the virus as a technical challenge, not realizing (or caring) that a virus attack can cost the collective victims as much as a hurricane or earthquake. Let us call our antihero Virgil the virus writer. If Virgil is typical, his goals are to produce a virus that spreads quickly, is difficult to detect, and is hard to get rid of once detected.

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. In addition, 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 here, so we will use the term "virus" to cover both for the moment. We will look at worms in Sec. 9.5.5.

9.5.1 Virus Damage Scenarios

Since a virus is just a program, it can do anything a program can do. For example, it can type a message, display an image on the screen, play music, or something else harmless. Unfortunately, it can also erase, modify, destroy, or steal files (by emailing them somewhere). Blackmail is also a possibility. Imagine a virus that encrypted all the files on the victim's hard disk, then displayed 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 thing a virus can do is render the computer unusable as long as the virus is running. This is called a **denial of service attack**. The usual approach is consume resources wildly, such as the CPU, or filling up the disk with junk. Here is a one-line program that used to wipe out any UNIX system:

```
main( ) {while (1) fork( );}
```

This program creates processes until the process table is full, preventing any other processes from starting. Now imagine a virus that infected every program in the system with this code. To guard against this problem, many modern UNIX systems limit the number of children a process may have at once.

Even worse, a virus can permanently damage the computer's hardware. Many modern computers hold the BIOS in flash ROM, which can be rewritten under program control (to allow the manufacturer to distribute bug fixes electronically). A virus can write random junk in the flash ROM so that the computer will no longer boot. If the flash ROM chip is in a socket, fixing the problem requires opening up the computer and replacing the chip. If the flash ROM chip is soldered to the parentboard, probably the whole board has to be thrown out and a new one purchased. Definitely not a fun experience.

A virus can also be released with a specific target. A company could release a virus 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 a targeted virus is one 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.

9.5.2 How Viruses Work

Enough for potential damage scenarios. Now let us see how viruses work. Virgil writes his virus, probably in assembly language, and then carefully inserts it into a program on his own machine using a tool called a **dropper**. That infected program is then distributed, perhaps by posting it to a bulletin board or 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. 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, 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 these viruses just overwrites the executable program with itself. These are called **overwriting viruses**. The infection logic of such a virus is given in Fig. 9-13.

```
#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 9-13. 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 safe keeping. Then it would tra-

verse 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. Nevertheless, it is interesting to note that the logic of Fig. 9-13, 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, most 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 of a program, it has to first copy the program to RAM, write itself at the front of the file, and then copy the program back from RAM following itself, as shown in Fig. 9-14(b). Unfortunately, the program will not run at its new virtual address, so either the virus has to relocate the program as it is moved, or slide it back to virtual address 0 after finishing its own execution.

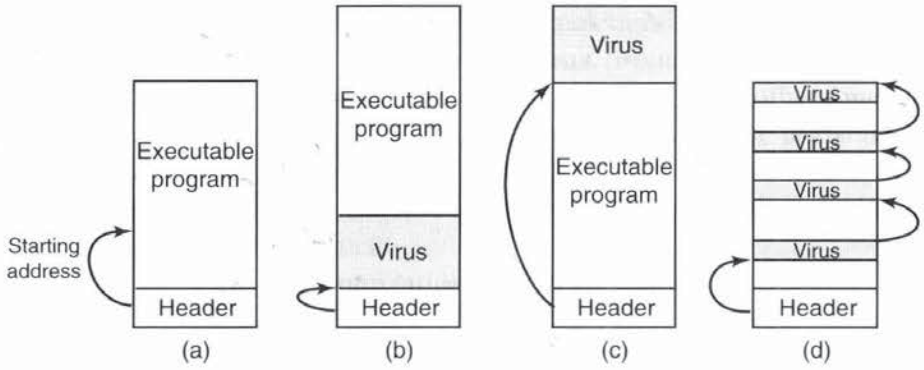


Figure 9-14. (a) An executable program. (b) With a virus at the front. (c) With a virus at the end. (d) With a virus spread over free space within the program.

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. 9-14(c). The virus will now execute at a different virtual address depending 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.

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. 9-14(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 exits. In contrast, a **memory-resident virus** stays in memory 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. 9-13 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 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 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. If the root directory is large enough and in a fixed place, as it is in Windows 98, the end of the root directory is also a possibility. 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 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.

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. It starts out by capturing all the interrupt vectors as shown in Fig. 9-15(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. 9-15(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. 9-15(c). When everything is loaded, the virus restores all the interrupt vectors and keeps only the system call trap vector for itself. After all, getting control on every system call is much more fun than getting control after every floppy disk operation, but during booting, it cannot take the risk of losing control forever. 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.

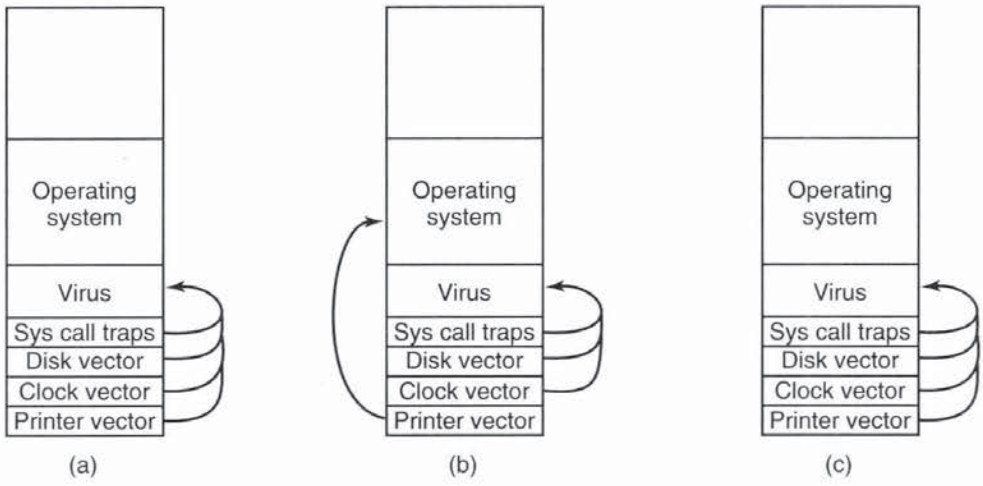


Figure 9-15. (a) After the virus has captured all the interrupt and trap vectors. (b) After the operating system has retaken the printer interrupt vector. (c) After the virus has noticed the loss of the printer interrupt vector and recaptured it.

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. 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 using a parasitic virus, 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.

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 only affects 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, sending documents with viruses embedded in macros is an immense problem. 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 the Macintosh, but not on UNIX). The most portable viruses of all are **source code viruses**. Imagine the virus of Fig. 9-13, 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 (for example, 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 only work 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. Virgil is an excellent programmer and has a lot of free time on his hands. Check your local newspaper for proof.

9.5.3 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 Web site. 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 hard 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.

In addition, the virus can check to see if there are any floppy disks in the drives, and if so, infect their files and boot sectors. Floppy disks are a good target because they get moved from machine to machine much more often than hard disks. If a floppy disk boot sector is infected and that disk is later used to boot a different machine, it can start infecting files and the hard disk boot sector on that machine. In the past, when floppy disks were the main transmission medium for programs, this mechanism was the main way viruses spread.

Nowadays, other options are available to Virgil. The virus can be written to check if the infected machine is on a LAN, something that is very likely on a machine belonging to a company or university. The virus can then start infecting unprotected files on the servers connected to this 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.

Often machines on a LAN have authorization to log onto remote machines over the Internet or a private corporate, 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 newsgroup or bulletin board system 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 these are probably hobbyists rather than career criminals, but the effects can be just as devastating. One other 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. If he posts the initial virus from his home machine he is running a certain risk. 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 floppy disk and read it in himself, or if the machines do not all have floppy disk drives, 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 two weeks 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 floppy disk and get the virus from a remote FTP site. Or bring a laptop and plug it in to an Ethernet or USB port that the Internet cafe has thoughtfully provided for laptop-toting tourists who want to read their email every day.

9.5.4 Antivirus and Anti-Antivirus Techniques

Viruses try to hide and users try to find them, which leads to a cat-and-mouse game. Let us now look at some of the issues here. To avoid showing up in directory listings, a companion virus, source code virus, or other file that should not be there can turn on the HIDDEN bit in Windows or use a file name beginning with the . character in UNIX. More sophisticated is to modify Windows' *explorer* or UNIX' *ls* to refrain from listing files whose names begin with *Virgil-*. Viruses can also hide in unusual and unsuspected places, such as the bad sector list on the disk or the Windows registry (an in-memory database available for programs to store uninterpreted strings). The flash ROM used to hold the BIOS and the CMOS memory are also possibilities although the former is hard to write and the latter is quite small. And, of course, the main workhorse of the virus world is infecting executable files and documents on the hard disk.

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 Web site 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, so 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, that is, warnings about legitimate files that 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. 9-16(a-b). However, a clever virus can avoid

detection by compressing the program and padding out the file to its original length. To make this scheme work, the virus must contain both compression and decompression procedures, as shown in Fig. 9-16(c).

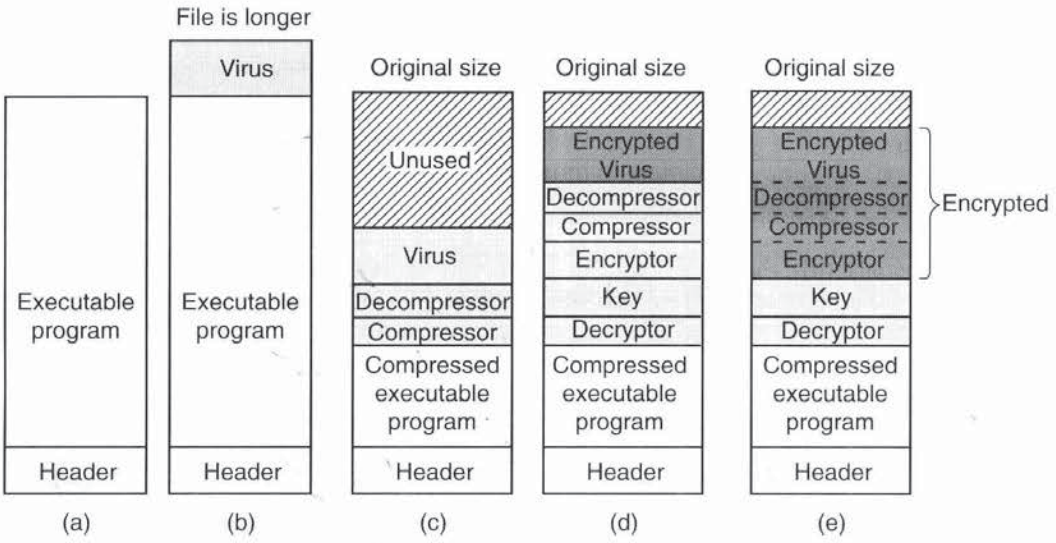


Figure 9-16. (a) A program. (b) An infected program. (c) A compressed infected program. (d) An encrypted virus. (e) A compressed virus with encrypted compression code.

Another way for the virus to try to escape detection is to make sure its representation on the disk does not look at all 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 with the contents of, say, 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. 9-16(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 procedure 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. 9-16(e). The decryption code cannot be encrypted, however. It has to actually execute on the hardware to decrypt the rest of the virus so it must be present in plaintext form. 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. 9-17(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. 9-17(b) does the same thing, only less efficiently due to the NOP (no operation) instructions interspersed with the real code.

```
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 9-17. Examples of a polymorphic virus.

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. 9-17(c) is functionally the same as the one of Fig. 9-17(a). When copying itself, the virus could use Fig. 9-17(c) instead of Fig. 9-17(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 during this piece of the code. Then Fig. 9-17(d) is also equivalent to Fig. 9-17(a). Finally, in many cases it is possible to swap instructions without changing what the program does, so we end up with Fig. 9-17(e) as another code fragment that is logically equivalent to Fig. 9-17(a). A piece of code that can mutate a sequence of machine instructions without changing its functionality is called a **mutation engine**, and sophisticated viruses contain them to mutate the decryptor from copy to copy. The mutation engine itself can be hidden by encrypting it along with the body of the virus.

Asking the poor antivirus software to realize that Fig. 9-17(a) through Fig. 9-17(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 ROM, CMOS memory, etc., 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 IDE, 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 as well.

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 and 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 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 encrypt it to make tampering easier to detect. Ideally, the encryption 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 ROM 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 *.doc* 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 interesting 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.

Second, install only shrink-wrapped 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 Web sites and bulletin boards 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 Web site.

Fourth, do not click on 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 floppy disk, CD-recordable, or tape. 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.

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. 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, flash ROM is a nice idea, but it should only be modifiable when an external toggle switch has been flipped, something that will only happen 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 hit the financial world and reset all bank accounts to 0. Of course, by then it would be too late.

Recovery from a Virus Attack

When a virus is detected, the computer should be halted immediately since a memory-resident virus may still be running. The computer should be rebooted from a CD-ROM or floppy disk that has always been write protected, and which contains the full operating system to bypass the boot sector, hard disk copy of the operating system, and disk drivers, all of which may now be infected. Then an antivirus program should be run from its original CD-ROM, since the hard disk version may also be infected.

The antivirus program may detect some viruses and may even be able to eliminate them, but there is no guarantee that it will get them all. Probably the safest

course of action at this point is to save all files that cannot contain viruses (like ASCII and JPEG files). Those files that might contain viruses (like *Word* files) should be converted to another format that cannot contain viruses, such as flat ASCII text (or at least the macros should be removed). All the saved files should be saved on an external medium. Then the hard disk should be reformatted using a format program taken from a write-protected floppy disk or a CD-ROM to insure that it itself is not infected. It is especially important that the MBR and boot sectors are also fully erased. Then the operating system should be reinstalled from the original CD-ROM. When dealing with virus infections, paranoia is your best friend.

9.5.5 The Internet Worm

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 (Spafford, 1989). For the story viewed as a police thriller, see (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. Working 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 BSD 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. At every BSD site 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. 9-11(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 wrote 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 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 1 in 7 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, the worm would probably have gone undetected.

Morris was caught when one of his friends spoke with the *New York Times* computer 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.

One permanent effect of this incident was the establishment of **CERT** (**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.

9.5.6 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 last year), 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 Web sites 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 Web site. 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, thus 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 that 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 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 three of these methods: sandboxing, interpretation, and code signing. Each one has its own strengths and weaknesses.

Sandboxing

The first method, called **sandboxing**, attempts to confine 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 all addresses within a sandbox had 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 for the case of 16 sandboxes of 16 MB each. Fig. 9-18(a),

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 rea-

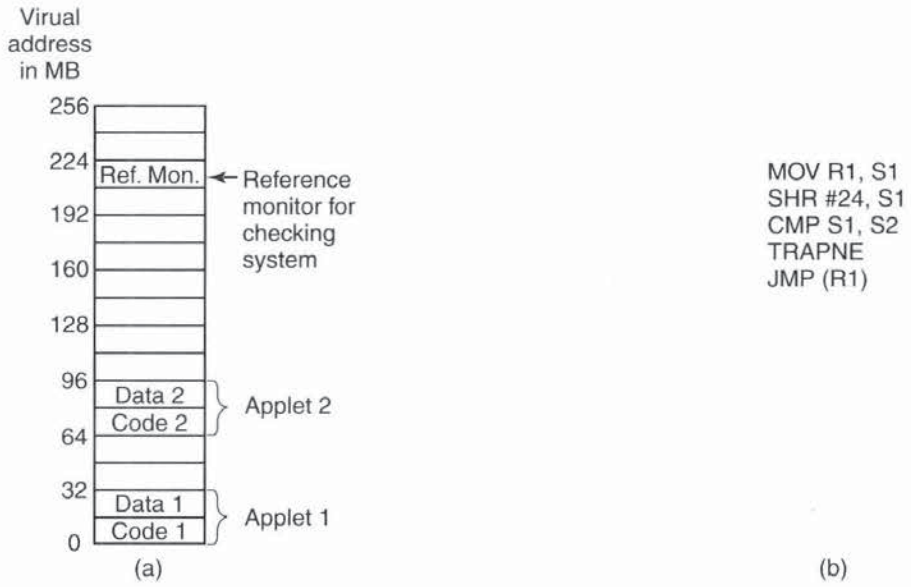


Figure 9-18. (a) Memory divided into 16-MB sandboxes. (b) One way of checking an instruction for validity.

son 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 JMPs. 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. 9-18(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. 9-19.

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 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

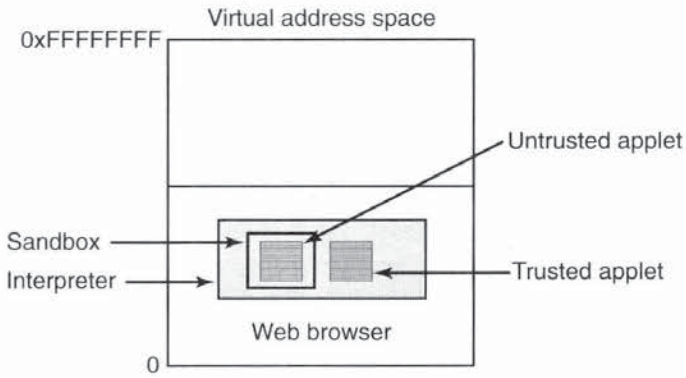


Figure 9-19. Applets can be interpreted by a Web browser.

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.

Code

Yet another way to deal with applet security is to know where they came from and only accept applets from trusted sources. With this approach, a user can maintain a list of trusted applet vendors and only run applets from those vendors. Applets from all other sources are rejected as too dicey. In this approach, no actual security mechanisms are present at run time. Applets from trustworthy vendors are run as is and code from other vendors is not run at all or in a restricted way (sandboxed or interpreted with little or no access to user files and other system resources).

To make this scheme work, as a minimum, there has to be a way for a user to determine that an applet was written by a trustworthy vendor and not modified by anyone after being produced. This is done using a digital signature, which allows the vendor to sign the applet in such a way that future modifications can be detected.

Code signing is based on public-key cryptography. An applet vendor, typically a software company, generates a (public key, private key) pair, making the former public and zealously guarding the latter. To sign an applet, the vendor first computes a hash function of the applet to get a 128-bit or 160-bit number, depending on whether MD5 or SHA is used. It then signs the hash value by encrypting it with its private key (actually, decrypting it using the notation of Fig. 9-3). This signature accompanies the applet wherever it goes.

When the user gets the applet, the browser computes the hash function itself. It then decrypts the accompanying signature using the vendor's public key and compares what the vendor claims the hash function is with what the browser itself computed. If they agree, the applet is accepted as genuine. Otherwise it is rejected as a forgery. The mathematics involved makes it exceedingly difficult for anyone to tamper with the applet in such a way as its hash function will match the hash function that is 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. 9-20.

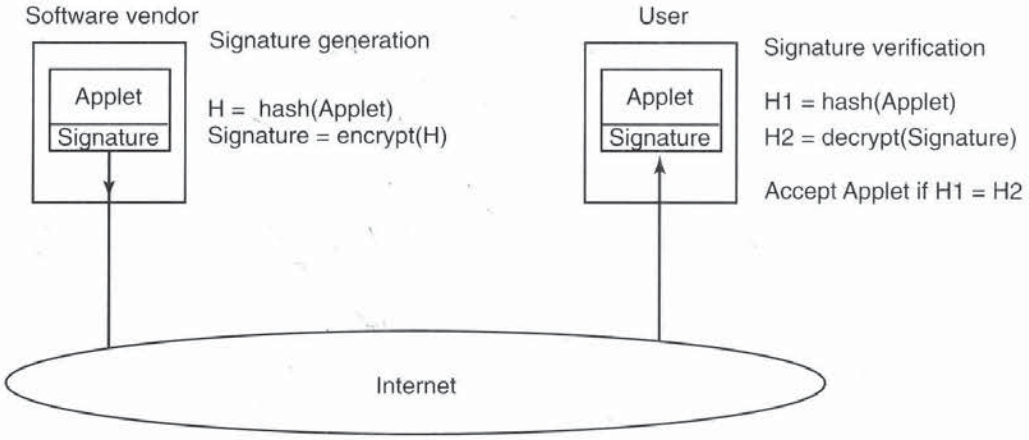


Figure 9-20. How code signing works.

9.5.7 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, 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 for remote execution are JVM programs.

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. 9-19, 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 provides 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 can be 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. A 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. 9-21, 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 9-21. 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. 9-21 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, *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 already 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.

9.6 PROTECTION MECHANISMS

In the previous sections we have looked at many potential problems, some of them technical, some of them not. In the following sections we will concentrate on some of the detailed technical ways that are used in operating systems to protect files and other things. All of these techniques make a clear distinction between policy (whose data are to be protected from whom) and mechanism (how the system enforces the policy). The separation of policy and mechanism is discussed in (Sandhu, 1993). Our emphasis will be on mechanisms, not policies.

In some systems, protection is enforced by a program called a **reference monitor**. Every time an access to a potentially protected resource is attempted, the system first asks the reference monitor to check its legality. The reference monitor then looks at its policy tables and makes a decision. Below we will describe the environment in which a reference monitor operates.

9.6.1 Protection Domains

A computer system contains many "objects" that need to be protected. These objects can be hardware (e.g., CPUs, memory segments, disk drives, or printers), or they can be 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.

Figure 9-22 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. Although not shown in this example, it is possible for the same object to be in multiple domains, with *different* rights in each one.

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. In UNIX, the domain of a process is defined by its UID and GID. Given any

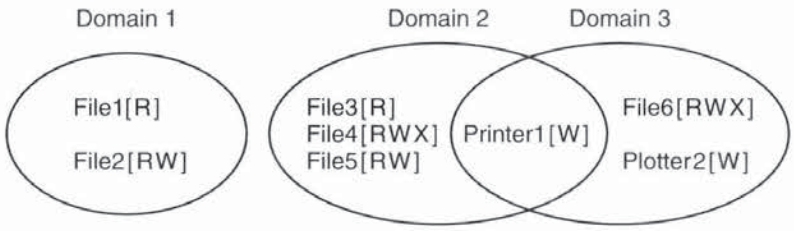


Figure 9-22. Three protection domains.

(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 in most cases.

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. 9-22 is shown in Fig. 9-23. 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							
Domain	File1	File2	File3	File4	File5	File6	Printer1	Plotter2
1	Read	Read Write						
2			Read	Read Write Execute	Read Write		Write	
3						Read Write Execute	Write	Write

Figure 9-23. 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 9-24 shows the matrix of Fig. 9-23 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 9-24. A protection matrix with domains as objects.

9.6.2 Access Control Lists

In practice, actually storing the matrix of Fig. 9-24 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 one 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 **Access Control List** or **ACL** and is illustrated in Fig. 9-25. 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

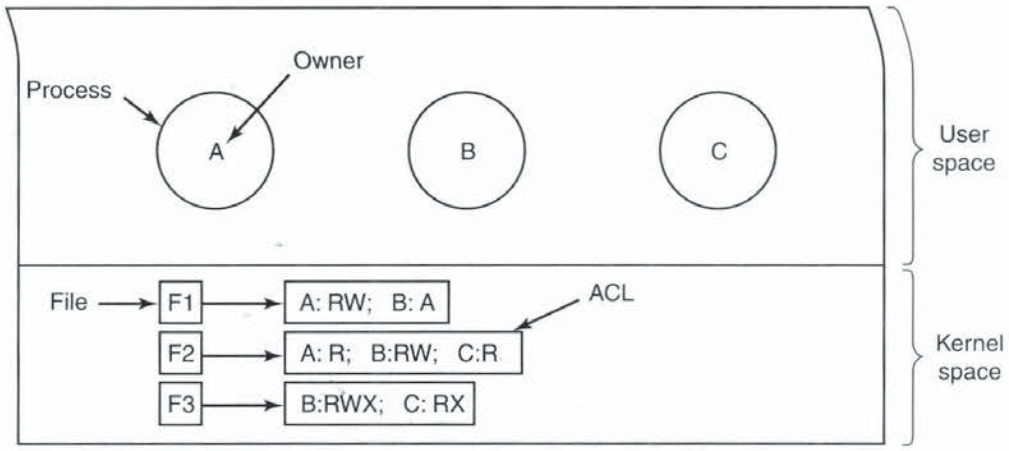


Figure 9-25. Use of access control lists to manage file access.

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 in addition *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.

This example illustrates the most basic form of protection with ACLs. More sophisticated systems are often used in practice. To start with, we have only shown 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 an 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. 9-26.

File	Access control list
Password	tana, sysadm: RW
Pigeon_data	bill, pigfan: RW; tana, pigfan: RW; ...

Figure 9-26. Two access control lists.

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 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 only do that 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 **wildcards**, which mean 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. In this case, 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 only take effect 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 at all.

9.6.3 Capabilities

The other way of slicing up the matrix of Fig. 9-24 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. 9-27.

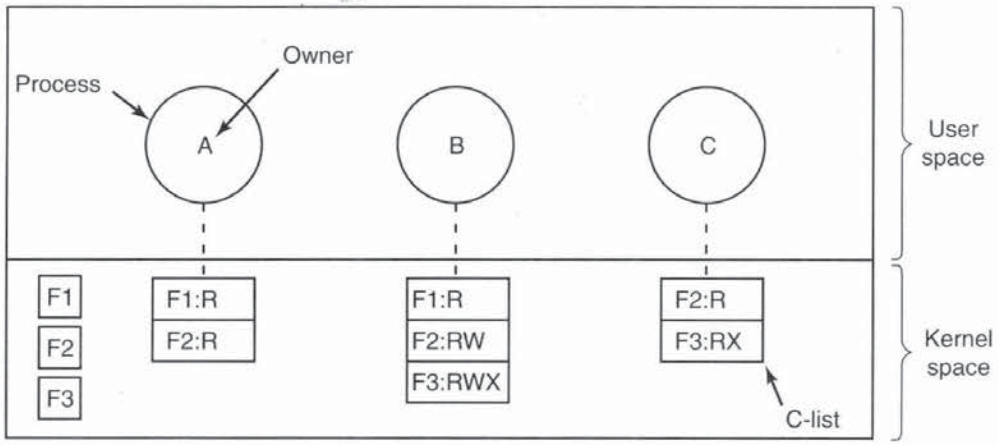


Figure 9-27. When capabilities are used, each process has a capability list.

Each capability grants the owner certain rights on a certain object. In Fig. 9-27, 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, etc. 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. 9-28.



Figure 9-28. 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. The last field consists of the concatenation of the object, rights, and check field run through a cryptographically-secure one-way function, f, of the kind we discussed earlier.

When the user wishes to access the object, it 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 one 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, if 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 and used extensively there (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 this problem, but this solution does not work well in a distributed system such as Amoeba.

On the other hand, capabilities solve the problem of sandboxing mobile code very elegantly. When a foreign program is started, it is given a capability list containing only those capabilities that the machine owner wants to give it, such as the ability to write on the screen and the ability to read and write files in one scratch directory just created for it. If the mobile code is put into its own process with only these limited capabilities, it will not be able to access any other system re-

sources and thus be effectively confined to a sandbox without the need to modify its code or run it interpretively. Running code with as few access rights as possible is known as the **principle of least privilege** and is a powerful guideline for producing secure systems.

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 the capabilities are removed and an object is not, problems arise. ACLs do not suffer from this problem.

9.7 TRUSTED SYSTEMS

Much of this chapter has been devoted to the fact that virtually all modern computer systems leak like a sieve. Estimates of the worldwide damage caused by viruses and similar problems exceed $1 trillion per year in wasted effort to repair problems, reconstruct damaged data, etc., not to mention lost business opportunities. 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 basically yes. How to build a secure system has been known for decades. MULTICS, designed in the 1960s, for example, had security as one of its main goals and achieved that fairly well.

Why secure systems are not being built is more complicated, but it 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 guaranteed to be immune 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.

The second issue is more subtle. The only way to build a secure system is to keep it simple. Features are the enemy of security. System designers believe (rightly or wrongly) that what users want is more features. More features mean more complexity, more code, more bugs, and more security errors.

Here are two simple examples. The first email systems sent messages as ASCII text. They were completely secure. There is nothing an incoming ASCII message can do to damage a computer system. 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 systems designers 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 (although illegal HTML could cause a buffer overflow attack). Now that many Web pages contain programs (applets) that the user has to run to view the content, one security leak after another pops up. As soon as one is fixed, another one takes its place. When the Web was entirely static, were users up in arms demanding dynamic content? Not that the author remembers, 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.

9.7.1 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 map 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. 9-29. 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.

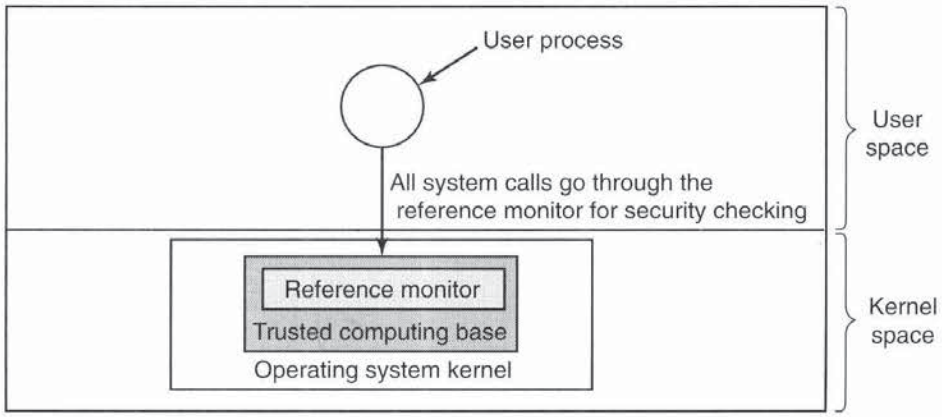


Figure 9-29. A reference monitor.

9.7.2 Formal Models of Secure Systems

Protection matrices, such as that of Fig. 9-23, 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. In the remainder of this section, we will 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. 9-30 in which domains correspond to users. In Fig. 9-30(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*.

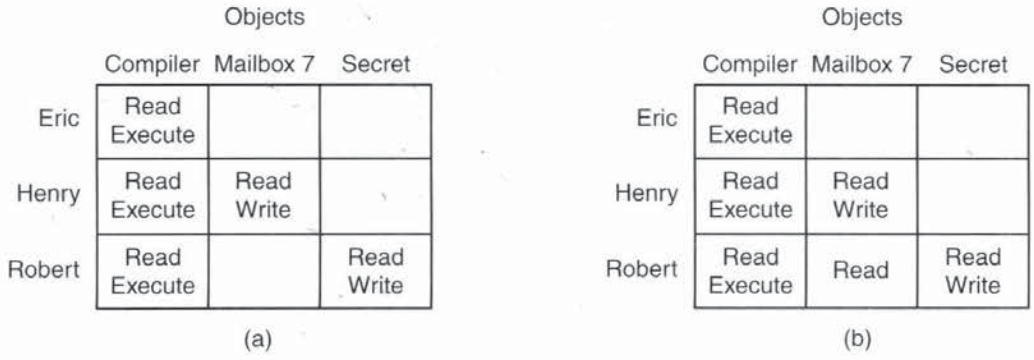


Figure 9-30. (a) An authorized state. (b) An unauthorized state.

Now imagine that *Robert* is very clever and has found a way to issue commands to have the matrix changed to Fig. 9-30(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. 9-30(b) is unauthorized.

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).

9.7.3 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. 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-La Padula Model

The most widely used multilevel security model is the **Bell-La Padula model** so we will start there (Bell and La Padula, 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-La Padula 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-La Padula model is illustrated graphically in Fig. 9-31.

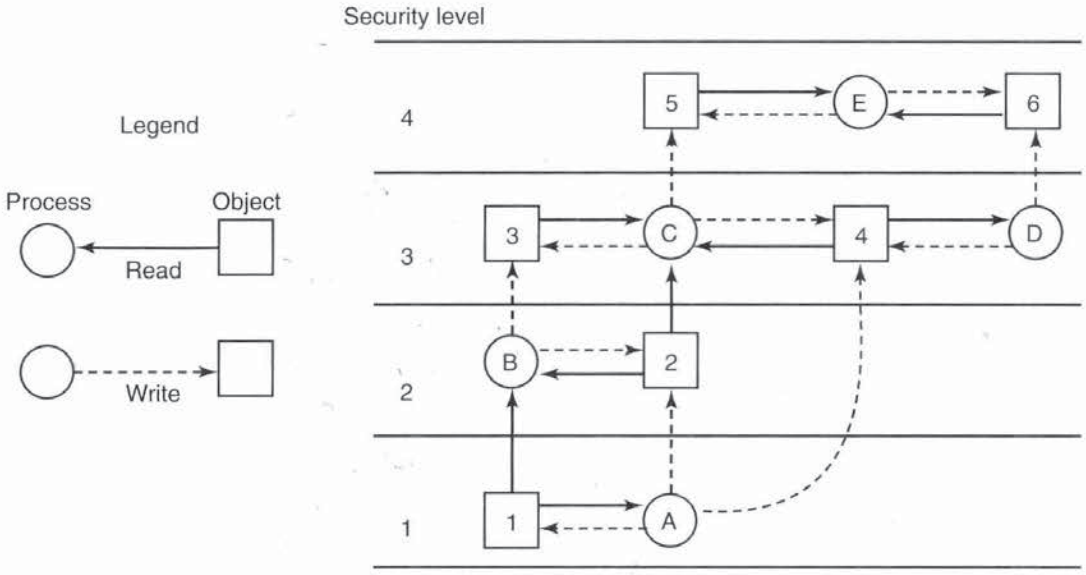


Figure 9-31. The Bell-La Padula multilevel security model.

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 up. The * property says that all dashed (write) arrows also go sideways or up. 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 a path that moves information downward, thus guaranteeing the security of the model.

The Biba Model

To summarize the Bell-La Padula 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-La Padula, 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-La Padula model is that it was devised to keep secrets, not guarantee the integrity of the data. To guarantee the integrity of the data, we need precisely the reverse properties (Biba, 1977):

1. **The simple integrity principle:** 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-La Padula properties and the Biba properties, but these are in direct conflict so they are hard to achieve simultaneously.

9.7.4 Orange Book Security

Given all this background, it should come as no surprise that the U.S. Dept. of Defense has put some effort into the area of secure systems. In particular, in 1985, it published a document formally known as Dept. of Defense standard DoD 5200.28, but usually called the **Orange Book** on account of its cover, which divides operating systems into seven categories based on their security properties. While the standard has since been replaced by another (and far more complex one), it is still a useful guide to some security properties. Also, one occasionally still sees vendor literature claiming conformance to some Orange Book security level. A table of the Orange Book requirements is given in Fig. 9-32. Below we will look at the security categories and point out some of the highlights.

Level D conformance is easy to achieve: it has no security requirements at all. It collects all the systems that have failed to pass even the minimum security tests. MS-DOS and Windows 95/98/Me are level D.

Level C is intended for environments with cooperating users. C1 requires a protected mode operating system, authenticated user login, and the ability for users to specify which files can be made available to other users and how (discretionary access control). Minimal security testing and documentation are also required.

C2 adds the requirement that discretionary access control is down to the level of the individual user. It also requires that objects (e.g., files, virtual memory pages) given to users must be initialized to all zeros, and a minimal amount of auditing is needed. The UNIX *rwx* scheme meets C1 but does not meet C2. For this, a more elaborate scheme, such as ACLs or equivalent, are needed.

Criterion	D	C1	C2	B1	B2	B3	A1
Security policy							
Discretionary access control		X	X	→	→	X	→
Object reuse			X	→	→	→	→
Labels				X	X	→	→
Label integrity				X	→	→	→
Exportation of labeled information				X	→	→	→
Labeling human readable output				X	→	→	→
Mandatory access control				X	X	→	→
Subject sensitivity labels					X	→	→
Device labels					X	→	→
Accountability							
Identification and authentication		X	X	X	→	→	→
Audit			X	X	X	X	→
Trusted path					X	X	→
Assurance							
System architecture		X	X	X	X	X	→
System integrity		X	→	→	→	→	→
Security testing		X	X	X	X	X	X
Design specification and verification				X	X	X	X
Covert channel analysis					X	X	X
Trusted facility management					X	X	→
Configuration management					X	→	X
Trusted recovery						X	→
Trusted distribution							X
Documentation							
Security features user's guide		X	→	→	→	→	→
Trusted facility manual		X	X	X	X	X	→
Test documentation		X	→	→	X	→	X
Design documentation		X	→	X	X	X	X

Figure 9-32. Orange Book security criteria. The symbol X means that there are new requirements here. The symbol → means that the requirements from the next lower category also apply here.

The B and A levels require all controlled users and objects to be assigned a security label, such as unclassified, secret, or top secret. The system must be capable of enforcing the Bell-La Padula information flow model.

B2 adds to this requirement that the system has been designed top-down in a modular way. The design must be presented in such a way that it can be verified. The possible covert channels (see the next section) must be analyzed.

B3 contains all of B2's features plus there must be ACLs with users and groups, a formal TCB must be presented, adequate security auditing must be present, and secure crash recovery must be included.

A1 requires a formal model of the protection system and a proof that the model is correct. It also requires a demonstration that the implementation conforms to the model. Covert channels must be formally analyzed.

9.7.5 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. In the purest form, it involves three processes on some protected machine. The first process is the client, which 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. 9-33. 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 be available. For example, the server can try to communicate a binary bit stream as follows. To send 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.

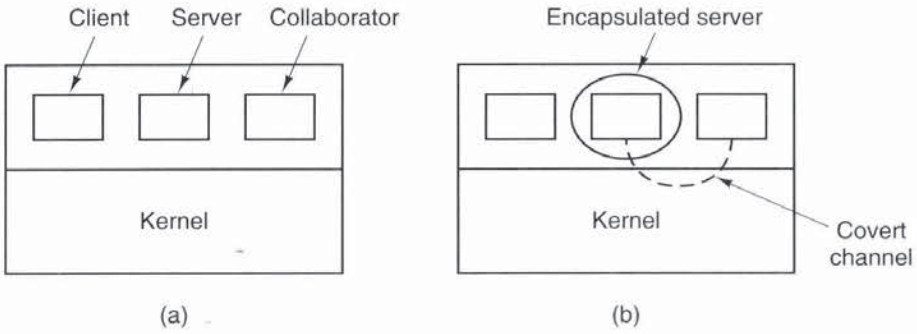


Figure 9-33. (a) The client, server, and collaborator processes. (b) The encapsulated server can still leak to the collaborator via covert channels.

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. 9-33(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. 9-34, 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 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

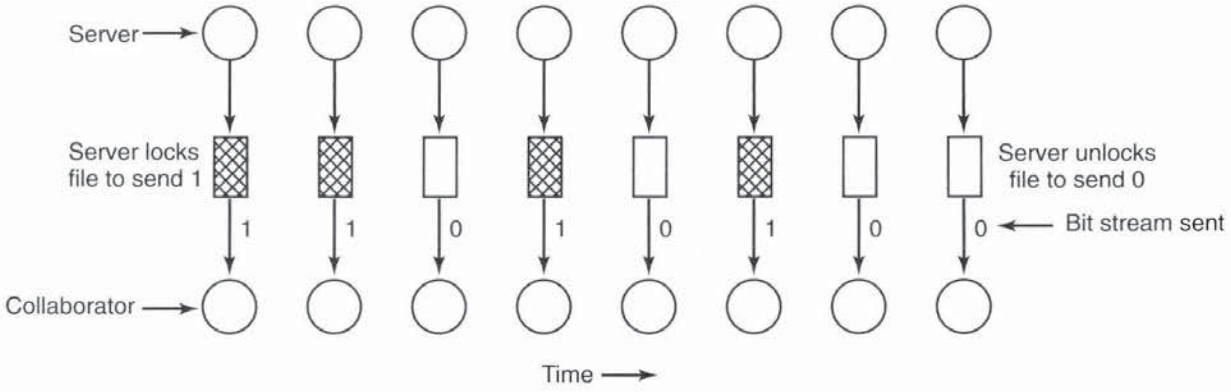


Figure 9-34. A covert channel using file locking.

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 dollars, the server could report the bill as $100.53 to its owner.

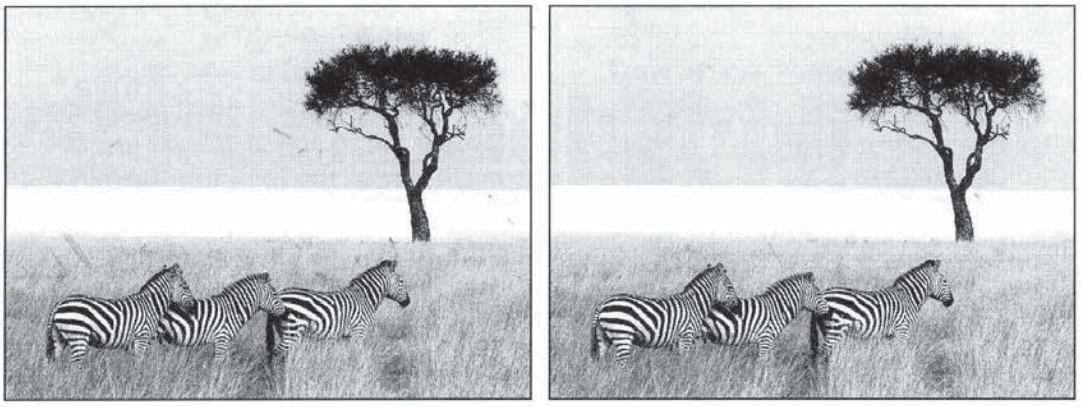
Just finding all the covert channels, let alone blocking them, is extremely difficult. In practice, there is little that can be done. Introducing a process that causes page faults at random, or otherwise spends its time degrading system performance in order to reduce the bandwidth of the covert channels is not an attractive proposition.

So far we have assumed that the client and server are separate processes. Another case is where there is only one process, the client, which is running a program containing a Trojan horse. The Trojan horse might have been written by the collaborator with the purpose of getting the user to run it in order to leak out data that the protection system prevents the collaborator from getting at directly.

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.

As a case in point, consider Fig. 9-35(a). This photograph, taken by the author in Kenya, contains three zebras contemplating an acacia tree. Fig. 9-35(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 9-35. (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 or 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 with governments 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, the author has prepared a demonstration, including the full-color image of Fig. 9-35(b) with the five plays embedded in it. The demonstration can be found at *www.cs.vu.nl/~ast/* . Click on the covered writing link 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.

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 2000, 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 errrors, 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.

9.8 RESEARCH ON SECURITY

Computer security is a very hot topic, with a great deal of research taking place, but most of it is not directly operating system related. Instead, it deals with network security (e.g., email, Web, and e-commerce security), cryptography, Java, or just managing a computer installation securely.

However, there is also some research closer to our subject. For example, user authentication is still important. Monrose and Rubin (1997) have studied it using keystroke dynamics, Pentland and Choudhury (2000) argue for face recognition, and Mark (2000) developed a way to model it, among others.

Some other operating systems related security work is as follows. Bershad et al. (1995a) have argued that protection is a software issue, not a hardware (i.e., MMU) issue. Mazieres et al. (1999) have looked at secure distributed file sys-

tems. Myers and Liskov (1997) studied secure information flow models. Chase et al. (1994) examined security in systems with a large address space occupied by multiple processes. Smart card security has been investigated by Clark and Hoffman (1994). Goldberg et al. (1998) have constructed virus phylogenies.

9.9 SUMMARY

Operating systems can be threatened in many ways, ranging from insider attacks to viruses coming in from the outside. Many attacks begin with a cracker trying to break into a specific system, often by just guessing passwords. These attacks often use dictionaries of common passwords and they are surprisingly successful. Password security can be made stronger using salt, one-time passwords, and challenge-response schemes. Smart cards and biometric indicators can also be used. Retinal scans are already practical.

Many different attacks on operating systems are known, including Trojan horse, login spoofing, logic bomb, trap door, and buffer overflow attacks. Generic attacks include asking for memory and snooping on it, making illegal system calls to see what happens, and even trying to trick insiders into revealing information they should not reveal.

Viruses are an increasingly serious problem for many users. Viruses come in many forms, including memory resident viruses, boot sector infectors, and macro viruses. Using a virus scanner to look for virus signatures is useful, but really good viruses can encrypt most of their code and modify the rest with each copy made, making detection very difficult. Some antivirus software works not by looking for specific virus signatures, but by looking for certain suspicious behavior. Avoiding viruses in the first place by safe computing practices is better than trying to deal with the aftermath of an attack. In short, do not load and execute programs whose origin is unknown and whose trustworthyness is questionable.

Mobile code is another issue that has to be dealt with these days. Putting it in a sandbox, interpreting it, and only running code signed by trusted vendors are possible approaches.

Systems can be protected using a matrix of protection domains (e.g., users) vertically and objects horizontally. The matrix can be sliced into rows, leading to capability-based systems or into columns, leading to ACL-based systems.

Secure systems can be designed, but that has to be a goal from the beginning. Probably the most important design rule is to have a minimal trusted computing base that cannot be bypassed whenever any resource is accessed. Multilevel security can be based on the Bell-La Padula model, designed for keeping secrets, or the Biba model, designed to maintaining system integrity. The Orange Book describes requirements trusted systems must meet. Finally, even if a system is provably secure, attention must be paid to covert channels, which can easily subvert the system by creating communication channels not included in the model.

PROBLEMS

1. Consider a secret-key cipher that has a 26×26 matrix with the columns headed by *ABC ... Z* and the rows are also *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?

2. Break the following monoalphabetic cipher. The plaintext, consisting of letters only, is a well-known excerpt from a poem by Lewis Carroll.

   kfd ktbd fzm eubd kfd pzyiom mztx ku kzyg ur bzha kfthcm
   ur mfudm zhx mftnm zhx mdzythc pzq ur ezsszcdm zhx gthcm
   zhx pfa kfd mdz tm sutythc fuk zhx pfdkfdi ntcm fzld pthcm
   sok pztk z stk kfd uamkdim eitdx sdruid pd fzld uoi efzk
   rui mubd ur om zid uok ur sidzkf zhx zyy ur om zid rzk
   hu foiia mztx kfd ezindhkdi kfda kfzhgdx ftb boef rui kfzk

3. 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?

4. 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 witht the receiver. How can the sender and receiver establish a new shared secret key under these circumstances?

5. Give a simple example of a mathematical function that to a first approximation will do as a one-way function.

6. 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 lower case 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?

7. 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 starting 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?

8. The Morris-Thompson protection scheme with the n-bit random numbers (salt) was designed to make it difficult for an intruder to discover a large number of passwords

by encrypting common strings in advance. Does the scheme also offer protection against a student user who is trying to guess the superuser password on his machine? Assume the password file is available for reading.

9. Name three characteristics that a good biometric indicator must have for it to be useful as a login authenticator.

10. Is there any feasible way to use the MMU hardware to prevent the kind of overflow attack shown in Fig. 9-11? Explain why or why not.

11. 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

```
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?

12. What property do the implementation of passwords in UNIX have in common with Lamport's scheme for logging in over an insecure network?

13. Lamport's one-time password scheme uses the passwords in reverse order. Would it not be simpler to use $f(s)$ the first time, $f(f(s))$ the second time, and so on?

14. As Internet cafes become more widespread, people are going to want ways of going to one anywhere in the world and conducting business from them. 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?

15. Can the Trojan horse attack work in a system protected by capabilities?

16. Name a C compiler feature that could eliminate a large number of security holes. Why is it not more widely implemented?

17. 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.

18. How could TENEX be modified to avoid the password problem described in the text?

19. 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?

20. Some operating systems require that disk partitions must start at the beginning of a track. How does this make life easier for a boot sector virus?

21. Change the program of Fig. 9-13 so it finds all the C programs instead of all the executable files.

22. The virus in Fig. 9-16(d) is encrypted. How can the dedicated scientists at the antivirus lab tell which part of the file is the key so they can decrypt the virus and reverse engineer it? What can Virgil do to make their job a lot harder?

23. The virus of Fig. 9-16(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?

24. Name one disadvantage of a polymorphic encrypting virus *from the point of view of the virus writer.*

25. Often one sees the following instructions for recovering from a virus attack:

1. Boot the infected system.
2. Back up all files to an external medium.
3. Run *fdisk* to format the disk.
4. Reinstall the operating system from the original CD-ROM.
5. Reload the files from the external medium.

Name two serious errors in these instructions.

26. Are companion viruses (viruses that do not modify any existing files) possible in UNIX? If so, how? If not, why not?

27. What is the difference between a virus and a worm? How do they each reproduce?

28. 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.

29. On some machines, the SHR instruction used in Fig. 9-18(b) fills the unused bits with zeros; on others the sign bit is extended to the right. For the correctness of Fig. 9-18(b), does it matter which kind of shift instruction is used? If so, which is better?

30. 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 of only *users*. Treat each of the two users and two groups as a domain, so 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

31. Express the permissions shown in the directory listing of the previous problem as access control lists.

32. Modify the ACL for one file to grant or deny an access that cannot be expressed using the UNIX *rwx* system. Explain this modification.

33. To verify that an applet has been signed by a trusted vendor, the applet vendor may include a certificate signed by 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?

34. 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?

35. 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.

36. 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 the friend can give it to yet someone else?

37. In Fig. 9-31, there is no arrow from process *B* to object *1*. Would such an arrow be allowed? If not, what rule would it violate?

38. In Fig. 9-31, there is no arrow from object *2* to process *A*. Would such an arrow be allowed? If not, what rule would it violate?

39. If process to process messages were allowed in Fig. 9-31, what rules would apply to them? For process *B* in particular, to which processes could it send messages and which not?

40. Consider the steganographic system of Fig. 9-35. Each pixel can be represented in a color space by a point in the 3-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.

41. 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 1600×1200 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 payload/(bytes transmitted)?

42. 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?

43. Write a pair of shell scripts to send and receive a text message by a covert channel on a UNIX system. (*Hint*: use the execution time of processes as your covert signal. The *sleep* command is guaranteed to run for a minimum time, set by its argument, and the *ps* command can be used to see all running processes).

44. 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 can 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.

10

CASE STUDY 1: UNIX AND LINUX

In the previous chapters, we examined many operating system principles, abstractions, algorithms, and techniques in general. Now it is time to look at some concrete systems to see how these principles are applied in the real world. We will begin with UNIX because it runs on a wider variety of computers than any other operating system. It is the dominant operating system on high-end workstations and servers, but it is also used on systems ranging from notebook computers to supercomputers. It was carefully designed with a clear goal in mind, and despite its age, is still modern and elegant. Many important design principles are illustrated by UNIX. Quite a few of these have been copied by other systems.

Our discussion of UNIX will start with its history and evolution of the system. Then we will provide an overview of the system, to give an idea of how it is used. This overview will be of special value to readers familiar only with Windows, since the latter hides virtually all the details of the system from its users. Although graphical interfaces may be easy for beginners, they provide little flexibility and no insight into how the system works.

Next we come to the heart of this chapter, an examination of processes, memory management, I/O, the file system, and security in UNIX. For each topic we will first discuss the fundamental concepts, then the system calls, and finally the implementation.

One problem that we will encounter is that there are many versions and clones of UNIX, including AIX, BSD, 1BSD, HP-UX, Linux, MINIX, OSF/1, SCO UNIX, System V, Solaris, XENIX, and various others, and each of these has gone through

many versions. Fortunately, the fundamental principles and system calls are pretty much the same for all of them (by design). Furthermore, the general implementation strategies, algorithms, and data structures are similar, but there are some differences. In this chapter we will draw upon several examples when discussing implementation, primarily 4.4BSD (which forms the basis for FreeBSD), System V Release 4, and Linux. Additional information about various implementations can be found in (Beck et al., 1998; Goodheart and Cox, 1994; Maxwell, 1999; McKusick et al., 1996; Pate, 1996; and Vahalia, 1996).

10.1 HISTORY OF UNIX

UNIX has a long and interesting history, so we will begin our study there. What started out as the pet project of one young researcher has become a multimillion dollar industry involving universities, multinational corporations, governments, and international standardization bodies. In the following pages we will tell how this story has unfolded.

10.1.1 UNICS

Back in the 1940s and 1950s, all computers were personal computers, at least in the sense that the then-normal way to use a computer was to sign up for an hour of time and take over the entire machine for that period. Of course, these machines were physically immense, but only one person (the programmer) could use them at any given time. When batch systems took over, in the 1960s, the programmer submitted a job on punched cards by bringing it to the machine room. When enough jobs had been assembled, the operator read them all in as a single batch. It usually took an hour or more after submitting a job until the output was returned. Under these circumstances, debugging was a time-consuming process, because a single misplaced comma might result in wasting several hours of the programmer's time.

To get around what almost everyone viewed as an unsatisfactory and unproductive arrangement, timesharing was invented at Dartmouth College and M.I.T. The Dartmouth system ran only BASIC and enjoyed a short-term commercial success before vanishing. The M.I.T. system, CTSS, was general purpose and was an enormous success among the scientific community. Within a short time, researchers at M.I.T. joined forces with Bell Labs and General Electric (then a computer vendor) and began designing a second generation system, **MULTICS** (**MULTiplexed Information and Computing Service**), as we discussed in Chap. 1.

Although Bell Labs was one of the founding partners in the MULTICS project, it later pulled out, which left one of the Bell Labs researchers, Ken Thompson, looking around for something interesting to do. He eventually decided to write a stripped down MULTICS by himself (in assembler this time) on a discarded PDP-7

minicomputer. Despite the tiny size of the PDP-7, Thompson's system actually worked and could support Thompson's development effort. Consequently, one of the other researchers at Bell Labs, Brian Kernighan, somewhat jokingly called it **UNICS** (**UNiplexed Information and Computing Service**). Despite puns about "EUNUCHS" being a castrated MULTICS, the name stuck, although the spelling was later changed to **UNIX**.

10.1.2 PDP-11 UNIX

Thompson's work so impressed his colleagues at Bell Labs, that he was soon joined by Dennis Ritchie, and later by his entire department. Two major developments occurred around this time. First, UNIX was moved from the obsolete PDP-7 to the much more modern PDP-11/20 and then later to the PDP-11/45 and PDP-11/70. The latter two machines dominated the minicomputer world for much of the 1970s. The PDP-11/45 and PDP-11/70 were powerful machines with large physical memories for their era (256 KB and 2 MB, respectively). Also, they had memory protection hardware, making it possible to support multiple users at the same time. However, they were both 16-bit machines that limited individual processes to 64 KB of instruction space and 64 KB of data space, even though the machine may have had far more physical memory.

The second development concerned the language in which UNIX was written. By now it was becoming painfully obvious that having to rewrite the entire system for each new machine was no fun at all, so Thompson decided to rewrite UNIX in a high-level language of his own design, called **B**. B was a simplified form of BCPL (which itself was a simplified form of CPL, which, like PL/I, never worked). Due to weaknesses in B, primarily lack of structures, this attempt was not successful. Ritchie then designed a successor to B, (naturally) called **C**, and wrote an excellent compiler for it. Together, Thompson and Ritchie rewrote UNIX in C. C was the right language at the right time, and has dominated system programming ever since.

In 1974, Ritchie and Thompson published a landmark paper about UNIX (Ritchie and Thompson, 1974). For the work described in this paper they were later given the prestigious ACM Turing Award (Ritchie, 1984; Thompson, 1984). The publication of this paper stimulated many universities to ask Bell Labs for a copy of UNIX. Since Bell Labs' parent company, AT&T, was a regulated monopoly at the time and was not permitted to be in the computer business, it had no objection to licensing UNIX to universities for a modest fee.

In one of those coincidences that often shape history, the PDP-11 was the computer of choice at nearly all university computer science departments, and the operating systems that came with the PDP-11 were widely regarded as being dreadful by professors and students alike. UNIX quickly filled the void, not in the least because it was supplied with the complete source code, so people could, and did, tinker with it endlessly. Numerous scientific meetings were organized around

UNIX, with distinguished speakers getting up in front of the room to tell about some obscure kernel bug they had found and fixed. An Australian professor, John Lions, wrote a commentary on the UNIX source code of the type normally reserved for the works of Chaucer or Shakespeare (reprinted as Lions, 1996). The book described Version 6, so named because it was described in the sixth edition of the UNIX Programmer's Manual. The source code was 8200 lines of C and 900 lines of assembly code. As a result of all this activity, new ideas and improvements to the system spread rapidly.

Within a few years, Version 6 was replaced by Version 7, the first portable version of UNIX (it ran on the PDP-11 and the Interdata 8/32), by now 18,800 lines of C and 2100 lines of assembler. A whole generation of students was brought up on Version 7, which contributed to its spread after they graduated and went to work in industry. By the mid 1980s, UNIX was in widespread use on minicomputers and engineering workstations from a variety of vendors. A number of companies even licensed the source code to make their own version of UNIX. One of these was a small startup called Microsoft, which sold Version 7 under the name XENIX for a number of years until its interest turned elsewhere.

10.1.3 Portable UNIX

Now that UNIX was written in C, moving it to a new machine, known as porting it, was much easier than in the early days. A port requires first writing a C compiler for the new machine. Then it requires writing device drivers for the new machine's I/O devices, such as terminals, printers, and disks. Although the driver code is in C, it cannot be moved to another machine, compiled, and run there because no two disks work the same way. Finally, a small amount of machine-dependent code, such as the interrupt handlers and memory management routines, must be rewritten, usually in assembly language.

The first port beyond the PDP-11 was to the Interdata 8/32 minicomputer. This exercise revealed a large number of assumptions that UNIX implicitly made about the machine it was running on, such as the unspoken supposition that integers held 16 bits, pointers also held 16 bits (implying a maximum program size of 64 KB), and that the machine had exactly three registers available for holding important variables. None of these were true on the Interdata, so considerable work was needed to clean UNIX up .

Another problem was that although Ritchie's compiler was fast and produced good object code, it produced only PDP-11 object code. Rather than write a new compiler specifically for the Interdata, Steve Johnson of Bell Labs designed and implemented the **portable C compiler**, which could be retargeted to produce code for any reasonable machine with a only a moderate amount of effort. For years, nearly all C compilers for machines other than the PDP-11 were based on Johnson's compiler, which greatly aided the spread of UNIX to new computers.

The port to the Interdata initially went slowly because all the development work had to be done on the only working UNIX machine, a PDP-11, which happened to be on the fifth floor at Bell Labs. The Interdata was on the first floor. Generating a new version meant compiling it on the fifth floor and then physically carrying a magnetic tape down to the first floor to see if it worked. After several months, a great deal of interest arose in the possibility of connecting these two machines together electronically. UNIX networking traces its roots to this link. After the Interdata port, UNIX was ported to the VAX and other computers.

After AT&T was broken up in 1984 by the U.S. government, the company was legally free to set up a computer subsidiary, and did. Shortly thereafter, AT&T released its first commercial UNIX product, System III. It was not well received, so it was replaced by an improved version, System V, a year later. Whatever happened to System IV is one of the great unsolved mysteries of computer science. The original System V has since been replaced by System V, releases 2, 3, and 4, each one bigger and more complicated than its predecessor. In the process, the original idea behind UNIX, of having a simple, elegant system has gradually diminished. Although Ritchie and Thompson's group later produced an 8th, 9th, and 10th edition of UNIX, these were never widely circulated, as AT&T put all its marketing muscle behind System V. However, some of the ideas from the 8th, 9th, and 10th editions were eventually incorporated into System V. AT&T eventually decided that it wanted to be a telephone company, not a computer company, after all, and sold its UNIX business to Novell in 1993. Novell then sold it to the Santa Cruz Operation in 1995. By then it was almost irrelevant who owned it since all the major computer companies already had licenses.

10.1.4 Berkeley UNIX

One of the many universities that acquired UNIX Version 6 early on was the University of California at Berkeley. Because the complete source code was available, Berkeley was able to modify the system substantially. Aided by grants from ARPA, the U.S. Dept. of Defense's Advanced Research Projects Agency, Berkeley produced and released an improved version for the PDP-11 called **1BSD** (**First Berkeley Software Distribution**). This tape was followed quickly by 2BSD also for the PDP-11.

More important were 3BSD and especially its successor, 4BSD, for the VAX. Although AT&T had a VAX version of UNIX, called **32V**, it was essentially Version 7. In contrast, 4BSD (including 4.1BSD, 4.2BSD, 4.3BSD, and 4.4BSD) contained a large number of improvements. Foremost among these was the use of virtual memory and paging, allowing programs to be larger than physical memory by paging parts of them in and out as needed. Another change allowed file names to be longer than 14 characters. The implementation of the file system was also changed, making it considerably faster. Signal handling was made more reliable.

Networking was introduced, causing the network protocol that was used, **TCP/IP**, to become a de facto standard in the UNIX world, and later in the Internet, which is dominated by UNIX-based servers.

Berkeley also added a substantial number of utility programs to UNIX, including a new editor (*vi*), a new shell (*csh*), Pascal and Lisp compilers, and many more. All these improvements caused Sun Microsystems, DEC, and other computer vendors to base their versions of UNIX on Berkeley UNIX, rather than on AT&T's "official" version, System V. As a consequence, Berkeley UNIX became well established in the academic, research, and defense worlds. For more information about Berkeley UNIX, see (McKusick et al., 1996).

10.1.5 Standard UNIX

By the late 1980s, two different, and somewhat incompatible, versions of UNIX were in widespread use: 4.3BSD and System V Release 3. In addition, virtually every vendor added its own nonstandard enhancements. This split in the UNIX world, together with the fact that there were no standards for binary program formats, greatly inhibited the commercial success of UNIX because it was impossible for software vendors to write and package UNIX programs with the expectation that they would run on any UNIX system (as was routinely done with MS-DOS). Various attempts at standardizing UNIX initially failed. AT&T, for example, issued the **SVID** (**System V Interface Definition**), which defined all the system calls, file formats, and so on. This document was an attempt to keep all the System V vendors in line, but it had no effect on the enemy (BSD) camp, which just ignored it.

The first serious attempt to reconcile the two flavors of UNIX was initiated under the auspices of the IEEE Standards Board, a highly respected and, most important, neutral body. Hundreds of people from industry, academia, and government took part in this work. The collective name for this project was **POSIX**. The first three letters refer to Portable Operating System. The *IX* was added to make the name UNIXish.

After a great deal of argument and counterargument, rebuttal and counterrebuttal, the POSIX committee produced a standard known as **1003.1**. It defines a set of library procedures that every conformant UNIX system must supply. Most of these procedures invoke a system call, but a few can be implemented outside the kernel. Typical procedures are *open*, *read*, and *fork*. The idea of POSIX is that a software vendor who writes a program that uses only the procedures defined by 1003.1 knows that this program will run on every conformant UNIX system.

While it is true that most standards bodies tend to produce a horrible compromise with a few of everyone's pet features in it, 1003.1 is remarkably good considering the large number of parties involved and their respective vested interests. Rather than take the union of all features in System V and BSD as the starting point (the norm for most standards bodies), the IEEE committee took the

intersection. Very roughly, if a feature was present in both System V and BSD, it was included in the standard; otherwise it was not. As a consequence of this algorithm, 1003.1 bears a strong resemblance to the direct ancestor of both System V and BSD, namely Version 7. The two areas in which it most strongly deviates from Version 7 are signals (which is largely taken from BSD) and terminal handling, which is new. The 1003.1 document is written in such a way that both operating system implementers and software writers can understand it, another novelty in the standards world, although work is already underway to remedy this.

Although the 1003.1 standard addresses only the system calls, related documents standardize threads, the utility programs, networking, and many other features of UNIX. In addition, the C language has also been standardized by ANSI and ISO.

Unfortunately, a funny thing happened on the way back from the standards meeting. Now that the System V versus BSD split had been dealt with, another one appeared. A group of vendors led by IBM, DEC, Hewlett-Packard, and many others did not like the idea that AT&T had control of the rest of UNIX, so they set up a consortium known as **OSF** (**Open Software Foundation**) to produce a system that met all the IEEE and other standards, but also contained a large number of additional features, such as a windowing system (X11), a graphical user interface (MOTIF), distributed computing (DCE), distributed management (DME), and much more.

AT&T's reaction was to set up its own consortium, **UI** (**UNIX International**) to do precisely the same thing. UI's version of UNIX was based on System V. The net result is that the world then had two powerful industry groups each offering their own version of UNIX, so the users were no closer to a standard than they were in the beginning. However, the marketplace decided that System V was a better bet than the OSF system, so the latter gradually vanished. Some companies have their own variants of UNIX, such as Sun's Solaris (based on System V).

10.1.6 MINIX

One property that all these systems have is that they are large and complicated, in a sense, the antithesis of the original idea behind UNIX. Even if the source code were freely available, which it is not in most cases, it is out of the question that a single person could understand it all any more. This situation led to the author of this book writing a new UNIX-like system that was small enough to understand, was available with all the source code, and could be used for educational purposes. That system consisted of 11,800 lines of C and 800 lines of assembly code. It was released in 1987, and was functionally almost equivalent to Version 7 UNIX, the mainstay of most computer science departments during the PDP-11 era.

MINIX was one of the first UNIX-like systems based on a microkernel design. The idea behind a microkernel is to provide minimal functionality in the kernel to

make it reliable and efficient. Consequently, memory management and the file system were pushed out into user processes. The kernel handled message passing between the processes and little else. The kernel was 1600 lines of C and 800 lines of assembler. For technical reasons relating to the 8088 architecture, the I/O device drivers (2900 additional lines of C) were also in the kernel. The file system (5100 lines of C) and memory manager (2200 lines of C) ran as two separate user processes.

Microkernels have the advantage over monolithic systems that they are easy to understand and maintain due to their highly modular structure. Also, moving code from the kernel to user mode makes them highly reliable because the crash of a user-mode process does less damage than the crash of a kernel-mode component. Their main disadvantage is a slightly lower performance due to the extra switches between user mode and kernel mode. However, performance is not everything: all modern UNIX systems run X Windows in user mode and simply accept the performance hit to get the greater modularity (in contrast to Windows, where the entire GUI is in the kernel). Other well-known microkernel designs of this era were Mach (Accetta et al., 1986) and Chorus (Rozier et al., 1988). A discussion of microkernel performance issues is given in (Bricker et al., 1991).

Within a few months of its appearance, MINIX became a bit of a cult item, with its own newsgroup, *comp.os.minix*, and over 40,000 users. Many users contributed commands and other user programs, so MINIX became a collective undertaking done by large numbers of users over the Internet. It was a prototype of other collaborative efforts that came later. In 1997, Version 2.0 of MINIX, was released and the base system, now including networking, had grown to 62,200 lines of code. A book about operating systems principles illustrated using the 500-page MINIX source code given in an appendix and on an accompanying CD-ROM is (Tanenbaum and Woodhull, 1997). MINIX is also available for free on the World Wide Web at URL *www.cs.vu.nl/~ast/minix.html* .

10.1.7 Linux

During the early years of MINIX development and discussion on the Internet, many people requested (or in many cases, demanded) more and better features, to which the author often said "No" (to keep the system small enough for students to understand completely in a one-semester university course). This continuous "No" irked many users. At this time, FreeBSD was not available, so that was not an option. After a number of years went by like this, a Finnish student, Linus Torvalds, decided to write another UNIX clone, named **Linux,** which would be a full-blown production system with many features MINIX was (intentionally) lacking. The first version of Linux, 0.01, was released in 1991. It was cross-developed on a MINIX machine and borrowed some ideas from MINIX ranging from the structure of the source tree to the layout of the file system. However, it was a monolithic rather than a microkernel design, with the entire operating sys-

tem in the kernel. The code size totaled 9,300 lines of C and 950 lines of assembler, roughly similar to MINIX version in size and also roughly comparable in functionality.

Linux rapidly grew in size and evolved into a full production UNIX clone as virtual memory, a more sophisticated file system, and many other features were added. Although it originally ran only on the 386 (and even had embedded 386 assembly code in the middle of C procedures), it was quickly ported to other platforms and now runs on a wide variety of machines, just as UNIX does. One difference with UNIX does stand out however: Linux makes use of many special features of the *gcc* compiler and would need a lot of work before it would compile with an ANSI standard C compiler with no extra features.

The next major release of Linux was version 1.0, issued in 1994. It was about 165,000 lines of code and included a new file system, memory-mapped files, and BSD-compatible networking with sockets and TCP/IP. It also included many new device drivers. Several minor revisions followed in the next two years.

By this time, Linux was sufficiently compatible with UNIX that a vast amount of UNIX software was ported to Linux, making it far more useful than it would have otherwise been. In addition, a large number of people were attracted to Linux and began working on the code and extending it in many ways under Torvalds' general supervision.

The next major release, 2.0, was made in 1996. It consisted of about 470,000 lines of C and 8000 lines of assembly code. It included support for 64-bit architectures, symmetric multiprogramming, new networking protocols, and numerous other features. A large fraction of the total code mass was taken up by an extensive collection of device drivers. Additional releases followed frequently.

A large array of standard UNIX software has been ported to Linux, including over 1000 utility programs, X Windows and a great deal of networking software. Two different GUIs (GNOME and KDE) have also been written for Linux. In short, it has grown to a full-blown UNIX clone with all the bells and whistles a UNIX lover might want.

One unusual feature of Linux is its business model: it is free software. It can be downloaded from various sites on the Internet, for example: *www.kernel.org*. Linux comes with a license devised by Richard Stallman, founder of the Free Software Foundation. Despite the fact that Linux is free, this license, the **GPL** (**GNU Public License**), is longer than Microsoft's Windows 2000 license and specifies what you can and cannot do with the code. Users may use, copy, modify, and redistribute the source and binary code freely. The main restriction is that all works derived from the Linux kernel may not be sold or redistributed in binary form only; the source code must either be shipped with the product or be made available on request.

Although Torvalds still controls the kernel fairly closely, a large amount of user-level software has been written by numerous other programmers, many of them originally migrated over from the MINIX, BSD, and GNU (Free Software

Foundation) online communities. However, as Linux evolves, a steadily smaller fraction of the Linux community want to hack source code (witness hundreds of books telling how to install and use Linux and only a handful discussing the code or how it works). Also, many Linux users now forego the free distribution on the Internet to buy one of many CD-ROM distributions available from numerous competing commercial companies. A Web site listing over 50 companies that sell different Linux packages is *www.linux.org*. As more and more software companies start selling their own versions of Linux and more and more hardware companies offer to preinstall it on the computers they ship, the line between commercial software and free software is beginning to blur a little.

As a footnote to the Linux story, it is interesting to note that just as the Linux bandwagon was gaining steam, it got a big boost from an unexpected source—AT&T. In 1992, Berkeley, by now running out of funding, decided to terminate BSD development with one final release, 4.4BSD, (which later formed the basis of FreeBSD). Since this version contained essentially no AT&T code, Berkeley issued the software under an open source license (not GPL) that let everybody do whatever they wanted with it except one thing—sue the University of California. The AT&T subsidiary controlling UNIX promptly reacted by—you guessed it—suing the University of California. It simultaneously sued a company, BSDI, set up by the BSD developers to package the system and sell support, much as Red Hat and other companies now do for Linux. Since virtually no AT&T code was involved, the lawsuit was based on copyright and trademark infringement, including items such as BSDI's 1-800-ITS-UNIX telephone number. Although the case was eventually settled out of court, this legal action kept FreeBSD off the market just long enough for Linux to get well established. Had the lawsuit not happened, starting around 1993 there would have been a serious competition between two free, open source UNIX systems: the reigning champion, BSD, a mature and stable system with a large academic following dating back to 1977 versus the vigorous young challenger, Linux, just two years old but with a growing following among individual users. Who knows how this battle of the free UNICES would have turned out?

Given this history, strict POSIX conformance, and overlap between the user communities, it should not come as a surprise that many of Linux' features, system calls, programs, libraries, algorithms, and internal data structures are very similar to those of UNIX. For example, over 80% of the ca. 150 Linux system calls are exact copies of the corresponding system calls in POSIX, BSD, or System V. Thus to a first approximation, much of the description of UNIX given in this chapter also applies to Linux. In places where there are substantial algorithmic differences between UNIX and Linux (e.g., the scheduling algorithm), we will point that out and cover both. Where they are essentially the same, for the sake of brevity we will just refer to UNIX. Be warned that Linux is evolving rapidly, and with thousands of people hacking away on it, some of this material (based on version 2.2) is sure to become obsolete before long.

10.2 OVERVIEW OF UNIX

In this section we will provide a general introduction to UNIX and how it is used, for the benefit of readers not already familiar with it. Although different versions of UNIX differ in subtle ways, the material presented here applies to all of them. The focus here is how UNIX appears at the terminal. Subsequent sections will focus on system calls and how it works inside.

10.2.1 UNIX Goals

UNIX is an interactive system designed to handle multiple processes and multiple users at the same time. It was designed by programmers, for programmers, to use in an environment in which the majority of the users are relatively sophisticated and are engaged in (often quite complex) software development projects. In many cases, a large number of programmers are actively cooperating to produce a single system, so UNIX has extensive facilities to allow people to work together and share information in controlled ways. The model of a group of experienced programmers working together closely to produce advanced software is obviously very different from the personal computer model of a single beginner working alone with a word processor, and this difference is reflected throughout UNIX from start to finish.

What is it that good programmers want in a system? To start with, most like their systems to be simple, elegant, and consistent. For example, at the lowest level, a file should just be a collection of bytes. Having different classes of files for sequential access, random access, keyed access, remote access, etc. (as mainframes do) just gets in the way. Similarly, if the command

```
ls A*
```

means list all the files beginning with "A" then the command

```
rm A*
```

should mean remove all the files beginning with "A" and not remove the one file whose name consists of an "A" and an asterisk. This characteristic is sometimes called the *principle of least surprise*.

Another thing that experienced programmers generally want is power and flexibility. This means that a system should have a small number of basic elements that can be combined in an infinite variety of ways to suit the application. One of the basic guidelines behind UNIX is that every program should do just one thing and do it well. Thus compilers do not produce listings, because other programs can do that better.

Finally, most programmers have a strong dislike for useless redundancy. Why type *copy* when *cp* is enough? To extract all the lines containing the string "ard" from the file *f*, the UNIX programmer types

```
grep ard f
```

The opposite approach is to have the programmer first select the *grep* program (with no arguments), and then have *grep* announce itself by saying: "Hi, I'm *grep*, I look for patterns in files. Please enter your pattern." After getting the pattern, *grep* prompts for a file name. Then it asks if there are any more file names. Finally, it summarizes what it is going to do and ask if that is correct. While this kind of user interface may or may not be suitable for rank novices, it irritates skilled programmers no end. What they want is a servant, not a nanny.

10.2.2 Interfaces to UNIX

A UNIX system can be regarded as a kind of pyramid, as illustrated in Fig. 10-1. At the bottom is the hardware, consisting of the CPU, memory, disks, terminals, and other devices. Running on the bare hardware is the UNIX operating system. Its function is to control the hardware and provide a system call interface to all the programs. These system calls allow user programs to create and manage processes, files, and other resources.

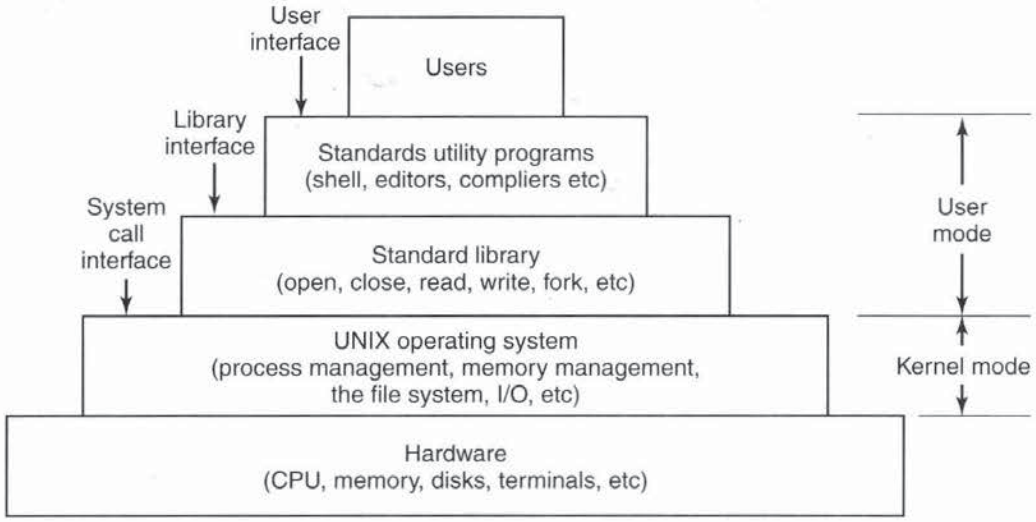


Figure 10-1. The layers in a UNIX system.

Programs make system calls by putting the arguments in registers (or sometimes, on the stack), and issuing trap instructions to switch from user mode to kernel mode to start up UNIX. Since there is no way to write a trap instruction in C, a library is provided, with one procedure per system call. These procedures are written in assembly language, but can be called from C. Each one first puts its arguments in the proper place, then executes the trap instruction. Thus to execute the read system call, a C program can call the *read* library procedure. As an aside, it is the library interface, and not the system call interface, that is specified by POSIX. In other words, POSIX tells which library procedures a conformant system must supply, what their parameters are, what they must do, and what results they must return. It does not even mention the actual system calls.

In addition to the operating system and system call library, all versions of UNIX supply a large number of standard programs, some of which are specified by the POSIX 1003.2 standard, and some of which differ between UNIX versions. These include the command processor (shell), compilers, editors, text processing programs, and file manipulation utilities. It is these programs that a user at a terminal invokes.

Thus we can speak of three different interfaces to UNIX: the true system call interface, the library interface, and the interface formed by the set of standard utility programs. While the latter is what the casual user thinks of as "UNIX," in fact, it has almost nothing to do with the operating system itself and can easily be replaced.

Some versions of UNIX, for example, have replaced this keyboard-oriented user interface with a mouse-oriented graphical user interface, without changing the operating system itself at all. It is precisely this flexibility that makes UNIX so popular and has allowed it to survive numerous changes in the underlying technology so well.

10.2.3 The UNIX Shell

Many UNIX systems have a graphical user interface of the kind made popular by the Macintosh and later Windows. However, real programmers still prefer a command line interface, called the **shell**. It is much faster to use, more powerful, easily extensible, and does not give the user RSI from having to use a mouse all the time. Below we will briefly describe the Bourne shell (*sh*). Since then, many new shells have been written (*ksh*, *bash*, etc.). Although UNIX fully supports a graphical environment (X Windows), even in this world many programmers simply make multiple console windows and act as if they have half a dozen ASCII terminals each running the shell.

When the shell starts up, it initializes itself, then types a **prompt** character, often a percent or dollar sign, on the screen and waits for the user to type a command line.

When the user types a command line, the shell extracts the first word from it, assumes it is the name of a program to be run, searches for this program, and if it finds it, runs the program. The shell then suspends itself until the program terminates, at which time it tries to read the next command. What is important here is simply the observation that the shell is an ordinary user program. All it needs is the ability to read from and write to the terminal, and the power to execute other programs.

Commands may take arguments, which are passed to the called program as character strings. For example, the command line

```
cp src dest
```

invokes the *cp* program with two arguments, *src* and *dest*. This program interprets

the first one to be the name of an existing file. It makes a copy of this file and calls the copy *dest*.

Not all arguments are file names. In

```
head –20 file
```

the first argument, *–20*, tells *head* to print the first 20 lines of *file*, instead of the default number of lines, 10. Arguments that control the operation of a command or specify an optional value are called **flags**, and by convention are indicated with a dash. The dash is required to avoid ambiguity, because the command

```
head 20 file
```

is perfectly legal, and tells *head* to first print the initial 10 lines of a file called *20*, and then print the initial 10 lines of a second file called *file*. Most UNIX commands accept multiple flags and arguments.

To make it easy to specify multiple file names, the shell accepts **magic characters**, sometimes called **wild cards**. An asterisk, for example, matches all possible strings, so

```
ls *.c
```

tells *ls* to list all the files whose name ends in *.c* If files named *x.c*, *y.c*, and *z.c* all exist, the above command is equivalent to typing

```
ls x.c y.c z.c
```

Another wild card is the question mark, which matches any one character. A list of characters inside square brackets selects any of them, so

```
ls [ape]*
```

lists all files beginning with "a", "p", or "e".

A program like the shell does not have to open the terminal in order to read from it or write to it. Instead, when it (or any other program) starts up, it automatically has access to a file called **standard input** (for reading), a file called **standard output** (for writing normal output), and a file called **standard error** (for writing error messages). Normally, all three default to the terminal, so that reads from standard input come from the keyboard and writes to standard output or standard error go to the screen. Many UNIX programs read from standard input and write to standard output as the default. For example,

```
sort
```

invokes the *sort* program, which reads lines from the terminal (until the user types a CTRL-D, to indicate end of file), sorts them alphabetically, and writes the result to the screen.

It is also possible to redirect standard input and standard output, as that is often useful. The syntax for redirecting standard input uses a less than sign (<)

followed by the input file name. Similarly, standard output is redirected using a greater than sign (>). It is permitted to redirect both in the same command. For example, the command

```
sort <in >out
```

causes *sort* to take its input from the file *in* and write its output to the file *out*. Since standard error has not been redirected, any error messages go to the screen. A program that reads its input from standard input, does some processing on it, and writes its output to standard output is called a **filter**.

Consider the following command line consisting of three separate commands:

```
sort <in >temp; head –30 <temp; rm temp
```

It first runs *sort*, taking the input from *in* and writing the output to *temp*. When that has been completed, the shell runs *head*, telling it to print the first 30 lines of *temp* and print them on standard output, which defaults to the terminal. Finally, the temporary file is removed.

It frequently occurs that the first program in a command line produces output that is used as the input on the next program. In the above example, we used the file *temp* to hold this output. However, UNIX provides a simpler construction to do the same thing. In

```
sort <in | head –30
```

the vertical bar, called the **pipe symbol**, says to take the output from *sort* and use it as the input to *head*, eliminating the need for creating, using, and removing the temporary file. A collection of commands connected by pipe symbols, called a **pipeline**, may contain arbitrarily many commands. A four-component pipeline is shown by the following example:

```
grep ter *.t | sort | head –20 | tail –5 >foo
```

Here all the lines containing the string "ter" in all the files ending in *.t* are written to standard output, where they are sorted. The first 20 of these are selected out by *head* which passes then to *tail*, which writes the last five (i.e., lines 16 to 20 in the sorted list) to *foo*. This is an example of how UNIX provides basic building blocks (numerous filters), each of which does one job, along with a mechanism for them to be put together in almost limitless ways.

UNIX is a general-purpose multiprogramming system. A single user can run several programs at once, each as a separate process. The shell syntax for running a process in the background is to follow its command with an ampersand. Thus

```
wc –l <a >b &
```

runs the word count program, *wc*, to count the number of lines (*–l* flag) in its input, *a*, writing the result to *b*, but does it in the background. As soon as the command has been typed, the shell types the prompt and is ready to accept and

handle the next command. Pipelines can also be put in the background, for example, by

```
sort <x | head &
```

Multiple pipelines can run in the background simultaneously.

It is possible to put a list of shell commands in a file and then start a shell with this file as standard input. The (second) shell just processes them in order, the same as it would with commands typed on the keyboard. Files containing shell commands are called **shell scripts**. Shell scripts may assign values to shell variables and then read them later. They may also have parameters, and use if, for, while, and case constructs. Thus a shell script is really a program written in shell language. The Berkeley C shell is an alternative shell that has been designed to make shell scripts (and the command language in general) look like C programs in many respects. Since the shell is just another user program, various other people have written and distributed a variety of other shells.

10.2.4 UNIX Utility Programs

The user interface to UNIX consists not only of the shell, but also of a large number of standard utility programs. Roughly speaking, these programs can be divided into six categories, as follows:

1. File and directory manipulation commands.
2. Filters.
3. Program development tools such as editors and compilers.
4. Text processing.
5. System administration.
6. Miscellaneous.

The POSIX 1003.2 standard specifies the syntax and semantics of just under 100 of these, primarily in the first three categories. The idea of standardizing them is to make it possible for anyone to write shell scripts that use these programs and work on all UNIX systems. In addition to these standard utilities, there are many application programs as well, of course, such as Web browsers, image viewers, etc.

Let us consider some examples of these programs, starting with file and directory manipulation.

```
cp a b
```

copies file *a* to *b*, leaving the original file intact. In contrast,

```
mv a b
```

copies *a* to *b* but removes the original. In effect, it moves the file rather than really making a copy in the usual sense. Several files can be concatenated using *cat*, which reads each of its input files and copies them all to standard output, one after another. Files can be removed by the *rm* command. The *chmod* command allows the owner to change the rights bits to modify access permissions. Directories can be created with *mkdir* and removed with *rmdir*. To see a list of the files in a directory, *ls* can be used. It has a vast number of flags to control how much detail about each file is shown (e.g., size, owner, group, creation date), to determine the sort order (e.g., alphabetical, by time of last modification, reversed), to specify the layout on the screen, and much more.

We have already seen several filters: *grep* extracts lines containing a given pattern from standard input or one or more input files; *sort* sorts its input and writes it on standard output; *head* extracts the initial lines of its input; *tail* extracts the final lines of its input. Other filters defined by 1003.2 are *cut* and *paste*, which allow columns of text to be cut and pasted into files; *od* which converts its (usually binary) input to ASCII text, in octal, decimal, or hexadecimal; *tr*, which does character translation (e.g., lower case to upper case), and *pr* which formats output for the printer, including options to include running heads, page numbers, and so on.

Compilers and programming tools include *cc*, which calls the C compiler, and *ar*, which collects library procedures into archive files.

Another important tool is *make*, which is used to maintain large programs whose source code consists of multiple files. Typically, some of these are **header files**, which contain type, variable, macro, and other declarations. Source files often include these using a special *include* directive. This way, two or more source files can share the same declarations. However, if a header file is modified, it is necessary to find all the source files that depend on it, and recompile them. The function of *make* is to keep track of which file depends on which header, and similar things, and arrange for all the necessary compilations to occur automatically. Nearly all UNIX programs, except the smallest ones, are set up to be compiled with *make*.

A selection of the POSIX utility programs is listed in Fig. 10-2, along with a short description of each one. All UNIX systems have these programs, and many more.

10.2.5 Kernel Structure

In Fig. 10-1 we saw the overall structure of a UNIX system. Now let us zoom in and look more closely at the kernel before examining the various parts. Showing the kernel structure is slightly tricky since there are many different versions of UNIX, but although the diagram of Fig. 10-3 describes 4.4BSD, it also applies to many other versions with perhaps small changes here and there.

Program	Typical use
cat	Concatenate multiple files to standard output
chmod	Change file protection mode
cp	Copy one or more files
cut	Cut columns of text from a file
grep	Search a file for some pattern
head	Extract the first lines of a file
ls	List directory
make	Compile files to build a binary
mkdir	Make a directory
od	Octal dump a file
paste	Paste columns of text into a file
pr	Format a file for printing
rm	Remove one or more files
rmdir	Remove a directory
sort	Sort a file of lines alphabetically
tail	Extract the last lines of a file
tr	Translate between character sets

Figure 10-2. A few of the common UNIX utility programs required by POSIX.

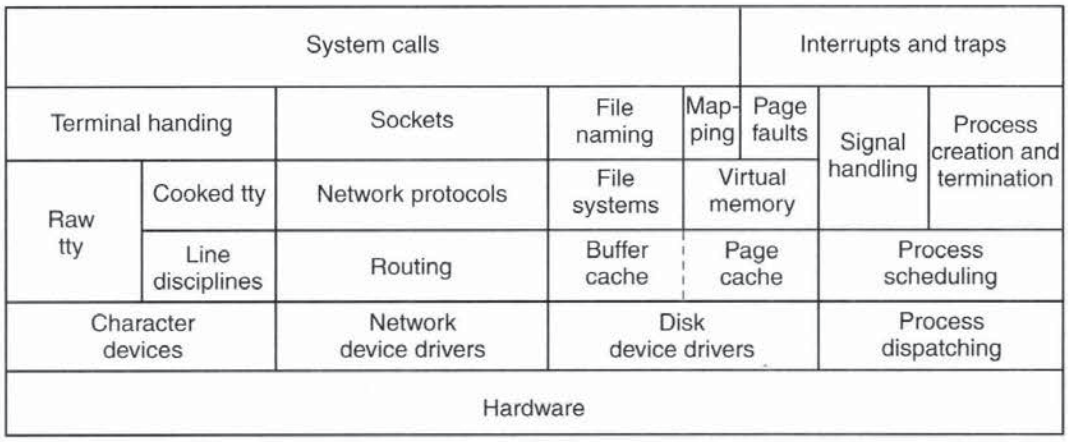


Figure 10-3. Structure of the 4.4BSD kernel.

The bottom layer of the kernel consists of the device drivers plus process dispatching. All UNIX drivers are classified as either character device drivers or block device drivers, with the main difference that seeks are allowed on block devices and not on character devices. Technically, network devices are character

devices, but they are handled so differently that it is probably clearer to separate them, as has been done in the figure. Process dispatching occurs when an interrupt happens. The low-level code here stops the running process, saves its state in the kernel process table, and starts the appropriate driver. Process dispatching also happens when the kernel is finished and it is time to start up a user process again. Dispatching code is in assembler and is quite distinct from scheduling.

Above the bottom level, the code is different in each of the four "columns" of Fig. 10-3. At the left, we have the character devices. There are two ways they are used. Some programs, such as visual editors like *vi* and *emacs*, want every key stroke as it is hit. Raw terminal (tty) I/O makes this possible. Other software, such as the shell (*sh*), is line oriented and allows users to edit the current line before hitting ENTER to send it to the program. This software uses cooked mode and line disciplines.

Networking software is often modular, with different devices and protocols supported. The layer above the network drivers handles a kind of routing function, making sure the right packet goes to the right device or protocol handler. Most UNIX systems contain the full functionality of an Internet router within the kernel, although the performance is less than that of a hardware router, but this code predated modern hardware routers. Above the router code is the actual protocol stack, always including IP and TCP, but sometimes additional protocols as well. Overlaying all the network is the socket interface, which allows programs to create sockets for particular networks and protocols, getting back a file descriptor for each socket to use later.

On top of the disk drivers are the file system's buffer cache and the page cache. In early UNIX systems, the buffer cache was a fixed chunk of memory, with the rest of memory for user pages. In many modern UNIX systems, there is no longer a fixed boundary, and any page of memory can be grabbed for either function, depending on what is needed more.

On top of the buffer cache come the file systems. Most UNIX systems support multiple file systems, including the Berkeley fast file system, log-structured file system, and various System V file systems. All of these file systems share the same buffer cache. On top of the file systems come file naming, directory management, hard link and symbolic link management, and other file system properties that are the same for all file systems.

On top of the page cache is the virtual memory system. All the paging logic is here, such as the page replacement algorithm. On top of it is the code for mapping files onto virtual memory and the high-level page fault management code. This is the code that figures out what to do when a page fault occurs. It first checks if the memory reference is valid, and if so, where the needed page is located and how it can be obtained.

The last column deals with process management. Above the dispatcher is the process scheduler, which chooses which process to run next. If threads are managed in the kernel, thread management is also here, although threads are managed

in user space on some UNIX systems. Above the scheduler comes the code for processing signals and sending them to the correct destination, as well as the process creation and termination code.

The top layer is the interface into the system. On the left is the system call interface. All system calls come here and are directed to one of the lower modules, depending on the nature of the call. On right part of the top layer is the entrance for traps and interrupts, including signals, page faults, processor exceptions of all kinds, and I/O interrupts.

10.3 PROCESSES IN UNIX

In the previous sections, we started out by looking at UNIX as viewed from the keyboard, that is, what the user sees at the terminal. We gave examples of shell commands and utility programs that are frequently used. We ended with a brief overview of the system structure. Now it is time to dig deeply into the kernel and look more closely at the basic concepts UNIX supports, namely, processes, memory, the file system, and input/output. These notions are important because the system calls—the interface to the operating system itself—manipulate them. For example, system calls exist to create processes, allocate memory, open files, and do I/O.

Unfortunately, with so many versions of UNIX in existence, there are some differences between them. In this chapter, we will emphasize the features common to all of them rather than focus on any one specific version. Thus in certain sections (especially implementation sections), the discussion may not apply equally to every version.

10.3.1 Fundamental Concepts

The only active entities in a UNIX system are the processes. UNIX processes are very similar to the classical sequential processes that we studied in Chap 2. Each process runs a single program and initially has a single thread of control. In other words, it has one program counter, which keeps track of the next instruction to be executed. Most versions of UNIX allow a process to create additional threads once it starts executing.

UNIX is a multiprogramming system, so multiple, independent processes may be running at the same time. Each user may have several active processes at once, so on a large system, there may be hundreds or even thousands of processes running. In fact, on most single-user workstations, even when the user is absent, dozens of background processes, called **daemons,** are running. These are started automatically when the system is booted. ("Daemon" is a variant spelling of "demon," which is a self-employed evil spirit.)

A typical daemon is the *cron daemon*. It wakes up once a minute to check if there is any work for it to do. If so, it does the work. Then it goes back to sleep until it is time for the next check.

This daemon is needed because it is possible in UNIX to schedule activities minutes, hours, days, or even months in the future. For example, suppose a user has a dentist appointment at 3 o'clock next Tuesday. He can make an entry in the cron daemon's database telling the daemon to beep at him at, say, 2:30. When the appointed day and time arrives, the cron daemon sees that it has work to do, and starts up the beeping program as a new process.

The cron daemon is also used to start up periodic activities, such as making daily disk backups at 4 A.M., or reminding forgetful users every year on October 31 to stock up on trick-or-treat goodies for Halloween. Other daemons handle incoming and outgoing electronic mail, manage the line printer queue, check if there are enough free pages in memory, and so forth. Daemons are straightforward to implement in UNIX because each one is a separate process, independent of all other processes.

Processes are created in UNIX in an especially simple manner. The fork system call creates an exact copy of the original process. The forking process is called the **parent process**. The new process is called the **child process**. The parent and child each have their own, private memory images. If the parent subsequently changes any of its variables, the changes are not visible to the child, and vice versa.

Open files are shared between parent and child. That is, if a certain file was open in the parent before the fork, it will continue to be open in both the parent and the child afterward. Changes made to the file by either one will be visible to the other. This behavior is only reasonable, because these changes are also visible to any unrelated process that opens the file as well.

The fact that the memory images, variables, registers, and everything else are identical in the parent and child leads to a small difficulty: How do the processes know which one should run the parent code and which one should run the child code? The secret is that the fork system call returns a 0 to the child and a nonzero value, the child's **PID** (**Process IDentifier**) to the parent. Both processes normally check the return value, and act accordingly, as shown in Fig. 10-4.

Processes are named by their PIDs. When a process is created, the parent is given the child's PID, as mentioned above. If the child wants to know its own PID, there is a system call, getpid, that provides it. PIDs are used in a variety of ways. For example, when a child terminates, the parent is given the PID of the child that just finished. This can be important because a parent may have many children. Since children may also have children, an original process can build up an entire tree of children, grandchildren, and further descendants.

Processes in UNIX can communicate with each other using a form of message passing. It is possible to create a channel between two processes into which one process can write a stream of bytes for the other to read. These channels are call-

```
pid = fork( );                  /* if the fork succeeds, pid > 0 in the parent */
if (pid < 0) {
     handle_error( );           /* fork failed (e.g., memory or some table is full) */
} else if (pid > 0) {
                                /* parent code goes here. /*/
} else {
                                /* child code goes here. /*/
}
```

Figure 10-4. Process creation in UNIX.

ed **pipes**. Synchronization is possible because when a process tries to read from an empty pipe it is blocked until data are available.

Shell pipelines are implemented with pipes. When the shell sees a line like

```
sort <f | head
```

it creates two processes, *sort* and *head*, and sets up a pipe between them in such a way that *sort*'s standard output is connected to *head*'s standard input. In this way, all the data that *sort* writes go directly to *head*, instead of going to a file. If the pipe fills up, the system stops running *sort* until *head* has removed some data from the pipe.

Processes can also communicate in another way: software interrupts. A process can send what is called a **signal** to another process. Processes can tell the system what they want to happen when a signal arrives. The choices are to ignore it, to catch it, or to let the signal kill the process (the default for most signals). If a process elects to catch signals sent to it, it must specify a signal handling procedure. When a signal arrives, control will abruptly switch to the handler. When the handler is finished and returns, control goes back to where it came from, analogous to hardware I/O interrupts. A process can only send signals to members of its **process group**, which consists of its parent (and further ancestors), siblings, and children (and further descendants). A process may also send a signal to all members of its process group with a single system call.

Signals are also used for other purposes. For example, if a process is doing floating-point arithmetic, and inadvertently divides by 0, it gets a a SIGFPE (floating-point exception) signal. The signals that are required by POSIX are listed in Fig. 10-5. Many UNIX systems have additional signals as well, but programs using them may not be portable to other versions of UNIX.

10.3.2 Process Management System Calls in UNIX

Let us now look at the UNIX system calls dealing with process management. The main ones are listed in Fig. 10-6. Fork is a good place to start the discussion. Fork is the only way to create a new process in UNIX systems. It creates an exact

Signal	Cause
SIGABRT	Sent to abort a process and force a core dump
SIGALRM	The alarm clock has gone off
SIGFPE	A floating-point error has occurred (e.g., division by 0)
SIGHUP	The phone line the process was using has been hung up
SIGILL	The user has hit the DEL key to interrupt the process
SIGQUIT	The user has hit the key requesting a core dump
SIGKILL	Sent to kill a process (cannot be caught or ignored)
SIGPIPE	The process has written to a pipe which has no readers
SIGSEGV	The process has referenced an invalid memory address
SIGTERM	Used to request that a process terminate gracefully
SIGUSR1	Available for application-defined purposes
SIGUSR2	Available for application-defined purposes

Figure 10-5. The signals required by POSIX.

duplicate of the original process, including all the file descriptors, registers and everything else. After the fork, the original process and the copy (the parent and child) go their separate ways. All the variables have identical values at the time of the fork, but since the entire parent core image is copied to create the child, subsequent changes in one of them do not affect the other one. The fork call returns a value, which is zero in the child, and equal to the child's PID in the parent. Using the returned PID, the two processes can see which is the parent and which is the child.

In most cases, after a fork, the child will need to execute different code from the parent. Consider the case of the shell. It reads a command from the terminal, forks off a child process, waits for the child to execute the command, and then reads the next command when the child terminates. To wait for the child to finish, the parent executes a waitpid system call, which just waits until the child terminates (any child if more than one exists). Waitpid has three parameters. The first one allows the caller to wait for a specific child. If it is −1, any old child (i.e., the first child to terminate) will do. The second parameter is the address of a variable that will be set to the child's exit status (normal or abnormal termination and exit value). The third one determines whether the caller blocks or returns if no child is already terminated.

In the case of the shell, the child process must execute the command typed by the user. It does this by using the exec system call, which causes its entire core image to be replaced by the file named in its first parameter. A highly simplified shell illustrating the use of fork, waitpid, and exec is shown in Fig. 10-7.

In the most general case, exec has three parameters: the name of the file to be executed, a pointer to the argument array, and a pointer to the environment array.

System call	Description
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, &statloc, opts)	Wait for a child to terminate
s = execve(name, argv, envp)	Replace a process' core image
exit(status)	Terminate process execution and return status
s = sigaction(sig, &act, &oldact)	Define action to take on signals
s = sigreturn(&context)	Return from a signal
s = sigprocmask(how, &set, &old)	Examine or change the signal mask
s = sigpending(set)	Get the set of blocked signals
s = sigsuspend(sigmask)	Replace the signal mask and suspend the process
s = kill(pid, sig)	Send a signal to a process
residual = alarm(seconds)	Set the alarm clock
s = pause( )	Suspend the caller until the next signal

Figure 10-6. Some system calls relating to processes. The return code *s* is −1 if an error has occurred, *pid* is a process ID, and *residual* is the remaining time in the previous alarm. The parameters are what the name suggests.

```
while (TRUE) {                              /* repeat forever /*/
     type_prompt( );                        /* display prompt on the screen */
     read_command(command, params);         /* read input line from keyboard */

     pid = fork( );                         /* fork off a child process */
     if (pid < 0) {
          printf("Unable to fork0);         /* error condition */
          continue;                         /* repeat the loop */
     }

     if (pid != 0) {
          waitpid (-1, &status, 0);         /* parent waits for child */
     } else {
          execve(command, params, 0);       /* child does the work */
     }
}
```

Figure 10-7. A highly simplified shell.

These will be described shortly. Various library procedures, including *execl*, *execv*, *execle*, and *execve*, are provided to allow the parameters to be omitted or specified in various ways. All of these procedures invoke the same underlying system call. Although the system call is exec, there is no library procedure with this name; one of the others must be used.

Let us consider the case of a command typed to the shell such as

```
cp file1 file2
```

used to copy *file1* to *file2*. After the shell has forked, the child locates and executes the file *cp* and passes it information about the files to be copied.

The main program of *cp* (and many other programs) contains the function declaration

```
main(argc, argv, envp)
```

where *argc* is a count of the number of items on the command line, including the program name. For the example above, *argc* is 3.

The second parameter, *argv*, is a pointer to an array. Element *i* of that array is a pointer to the *i*-th string on the command line. In our example, *argv*[0] would point to the string "cp". Similarly, *argv*[1] would point to the 5-character string "file1" and *argv*[2] would point to the 5-character string "file2".

The third parameter of *main*, *envp*, is a pointer to the environment, an array of strings containing assignments of the form *name* = *value* used to pass information such as the terminal type and home directory name to a program. In Fig. 10-7, no environment is passed to the child, so the third parameter of *execve* is a zero in this case.

If exec seems complicated, do not despair; it is the most complex system call. All the rest are much simpler. As an example of a simple one, consider exit, which processes should use when they are finished executing. It has one parameter, the exit status (0 to 255), which is returned to the parent in the variable *status* of the waitpid system call. The low-order byte of *status* contains the termination status, with 0 being normal termination and the other values being various error conditions. The high-order byte contains the child's exit status (0 to 255), as specified in the child's call to exit. For example, if a parent process executes the statement

```
n = waitpid(-1, &status, 0);
```

it will be suspended until some child process terminates. If the child exits with, say, 4 as the parameter to *exit*, the parent will be awakened with *n* set to the child's PID and *status* set to 0x0400 (0x as a prefix means hexadecimal in C). The low-order byte of *status* relates to signals; the next one is the value the child returned in its call to exit.

If a process exits and its parent has not yet waited for it, the process enters a kind of suspended animation called the **zombie state**. When the parent finally waits for it, the process terminates.

Several system calls relate to signals, which are used in a variety of ways. For example, if a user accidently tells a text editor to display the entire contents of a very long file, and then realizes the error, some way is needed to interrupt the editor. The usual choice is for the user to hit some special key (e.g., DEL or

CTRL-C), which sends a signal to the editor. The editor catches the signal and stops the print-out.

To announce its willingness to catch this (or any other) signal, the process can use the sigaction system call. The first parameter is the signal to be caught (see Fig. 10-5). The second is a pointer to a structure giving a pointer to the signal handling procedure, as well as some other bits and flags. The third one points to a structure where the system returns information about signal handling currently in effect, in case it must be restored later.

The signal handler may run for as long as it wants to. In practice, though, signal handlers are usually fairly short. When the signal handling procedure is done, it returns to the point from which it was interrupted.

The sigaction system call can also be used to cause a signal to be ignored, or to restore the default action, which is killing the process.

Hitting the DEL key is not the only way to send a signal. The kill system call allows a process to signal another related process. The choice of the name "kill" for this system call is not an especially good one, since most processes send signals to other ones with the intention that they be caught.

For many real-time applications, a process needs to be interrupted after a specific time interval to do something, such as to retransmit a potentially lost packet over an unreliable communication line. To handle this situation, the alarm system call has been provided. The parameter specifies an interval, in seconds, after which a SIGALRM signal is sent to the process. A process may have only one alarm outstanding at any instant. If an alarm call is made with a parameter of 10 seconds, and then 3 seconds later another alarm call is made with a parameter of 20 seconds, only one signal will be generated, 20 seconds after the second call. The first signal is canceled by the second call to alarm. If the parameter to alarm is zero, any pending alarm signal is canceled. If an alarm signal is not caught, the default action is taken and the signaled process is killed. Technically, alarm signals may be ignored, but that is a pointless thing to do.

It sometimes occurs that a process has nothing to do until a signal arrives. For example, consider a computer-aided instruction program that is testing reading speed and comprehension. It displays some text on the screen and then calls alarm to signal it after 30 seconds. While the student is reading the text, the program has nothing to do. It could sit in a tight loop doing nothing, but that would waste CPU time that a background process or other user might need. A better solution is to use the pause system call, which tells UNIX to suspend the process until the next signal arrives.

Thread Management System Calls

The first versions of UNIX did not have threads. That feature was added many years later. Initially there were many threads packages in use, but the pro-