stop transmitting. Each site will try again after some random time interval. The main problem with this approach is that, when the system is very busy, many collisions may occur, and thus performance may be degraded. Nevertheless, CSMA/CD has been used successfully in the Ethernet system, the most common local area network system. One strategy for limiting the number of collisions is to limit the number of hosts per Ethernet network. Adding more hosts to a congested network could result in poor network throughput. As systems get faster, they are able to send more packets per time segment. As a result, the number of systems per Ethernet network generally is decreasing so that networking performance is kept reasonable.

- **Token passing**. A unique message type, known as a **token**, continuously circulates in the system (usually a ring structure). A site that wants to transmit information must wait until the token arrives. It removes the token from the ring and begins to transmit its messages. When the site completes its round of message passing, it retransmits the token. This action, in turn, allows another site to receive and remove the token and to start its message transmission. If the token gets lost, the system must then detect the loss and generate a new token. It usually does that by declaring an **election** to choose a unique site where a new token will be generated. Later, in Section 18.6, we present one election algorithm. A token-passing scheme has been adopted by the IBM and HP/Apollo systems. The benefit of a token-passing network is that performance is constant. Adding new sites to a network may lengthen the waiting time for a token, but it will not cause a large performance decrease, as may happen on Ethernet. On lightly loaded networks, however, Ethernet is more efficient, because systems can send messages at any time.

16.6 Communication Protocols

When we are designing a communication network, we must deal with the inherent complexity of coordinating asynchronous operations communicating in a potentially slow and error-prone environment. In addition, the systems on the network must agree on a protocol or a set of protocols for determining host names, locating hosts on the network, establishing connections, and so on. We can simplify the design problem (and related implementation) by partitioning the problem into multiple layers. Each layer on one system communicates with the equivalent layer on other systems. Typically, each layer has its own protocols, and communication takes place between peer layers using a specific protocol. The protocols may be implemented in hardware or software. For instance, Figure 16.6 shows the logical communications between two computers, with the three lowest-level layers implemented in hardware. Following the International Standards Organization (ISO), we refer to the layers as follows:

1. **Physical layer**. The physical layer is responsible for handling both the mechanical and the electrical details of the physical transmission of a bit stream. At the physical layer, the communicating systems must agree on the electrical representation of a binary 0 and 1, so that when data are

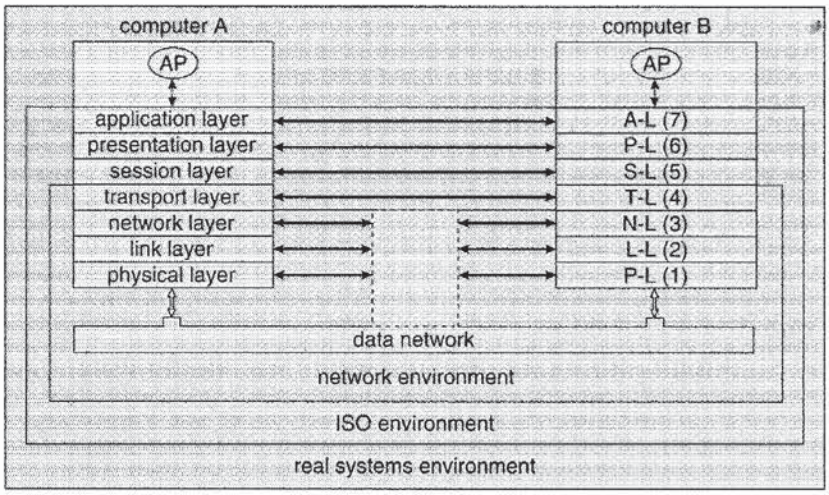


Figure 16.6 Two computers communicating via the ISO network model.

sent as a stream of electrical signals, the receiver is able to interpret the data properly as binary data. This layer is implemented in the hardware of the networking device.

2. **Data-link layer**. The data-link layer is responsible for handling *frames*, or fixed-length parts of packets, including any error detection and recovery that occurred in the physical layer.
3. **Network layer**. The network layer is responsible for providing connections and for routing packets in the communication network, including handling the addresses of outgoing packets, decoding the addresses of incoming packets, and maintaining routing information for proper response to changing load levels. Routers work at this layer.
4. **Transport layer**. The transport layer is responsible for low-level access to the network and for transfer of messages between clients, including partitioning messages into packets, maintaining packet order, controlling flow, and generating physical addresses.
5. **Session layer**. The session layer is responsible for implementing sessions, or process-to-process communication protocols. Typically, these protocols are the actual communications for remote logins and for file and mail transfers.
6. **Presentation layer**. The presentation layer is responsible for resolving the differences in formats among the various sites in the network, including character conversions and half duplex–full duplex modes (character echoing).
7. **Application layer**. The application layer is responsible for interacting directly with users. This layer deals with file transfer, remote-login protocols, and electronic mail, as well as with schemas for distributed databases.

Figure 16.7 summarizes the **ISO protocol stack**—a set of cooperating protocols—showing the physical flow of data. As mentioned, logically each layer of a protocol stack communicates with the equivalent layer on other systems. But physically, a message starts at or above the application layer and is passed through each lower level in turn. Each layer may modify the message and include message-header data for the equivalent layer on the receiving side. Ultimately, the message reaches the data-network layer and is transferred as one or more packets (Figure 16.8). The data-link layer of the target system receives these data, and the message is moved up through the protocol stack; it is analyzed, modified, and stripped of headers as it progresses. It finally reaches the application layer for use by the receiving process.

The ISO model formalizes some of the earlier work done in network protocols but was developed in the late 1970s and is currently not in widespread use. Perhaps the most widely adopted protocol stack is the TCP/IP model, which

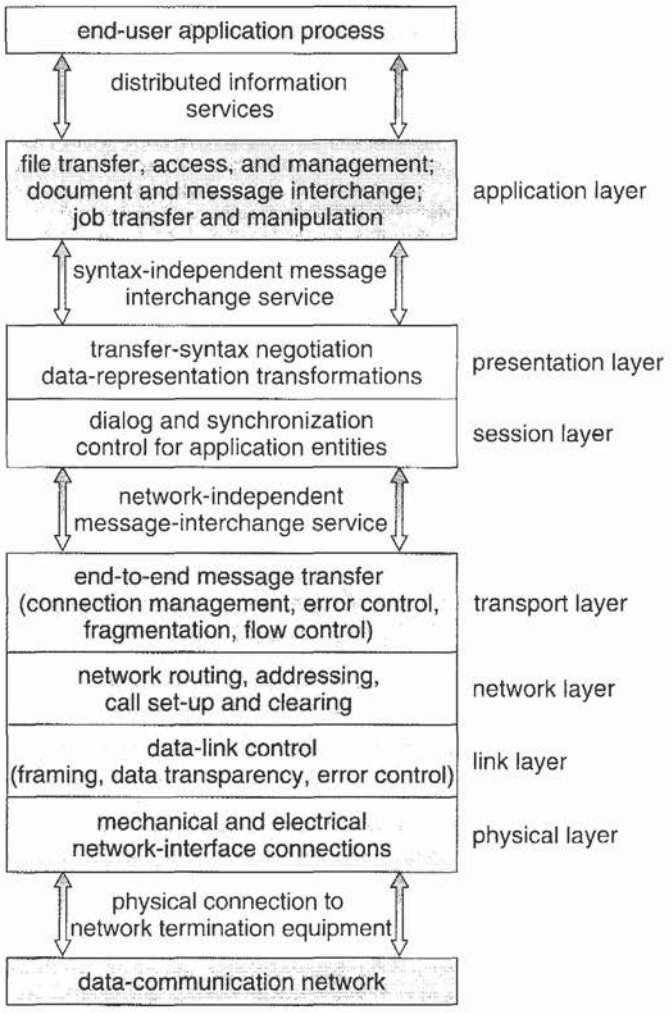


Figure 16.7 The ISO protocol stack.

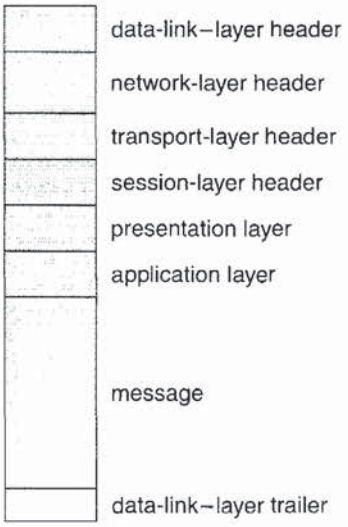


Figure 16.8 An ISO network message.

has been adopted by virtually all Internet sites. The TCP/IP protocol stack has fewer layers than does the ISO model. Theoretically, because it combines several functions in each layer, it is more difficult to implement but more efficient than ISO networking. The relationship between the ISO and TCP/IP models is shown in Figure 16.9. The TCP/IP application layer identifies several protocols in widespread use in the Internet, including HTTP, FTP, Telnet, DNS, and SMTP. The transport layer identifies the unreliable, connectionless **user datagram protocol** (**UDP**) and the reliable, connection-oriented **transmission control protocol** (**TCP**). The Internet protocol (**IP**) is responsible for routing IP datagrams through the Internet. The TCP/IP model does not formally identify a link or physical layer, allowing TCP/IP traffic to run across any physical network. In Section 16.9, we consider the TCP/IP model running over an Ethernet network.

16.7 Robustness

A distributed system may suffer from various types of hardware failure. The failure of a link, the failure of a site, and the loss of a message are the most common types. To ensure that the system is robust, we must detect any of these failures, reconfigure the system so that computation can continue, and recover when a site or a link is repaired.

16.7.1 Failure Detection

In an environment with no shared memory, we are generally unable to differentiate among link failure, site failure, and message loss. We can usually detect only that one of these failures has occurred. Once a failure has been detected, appropriate action must be taken. What action is appropriate depends on the particular application.

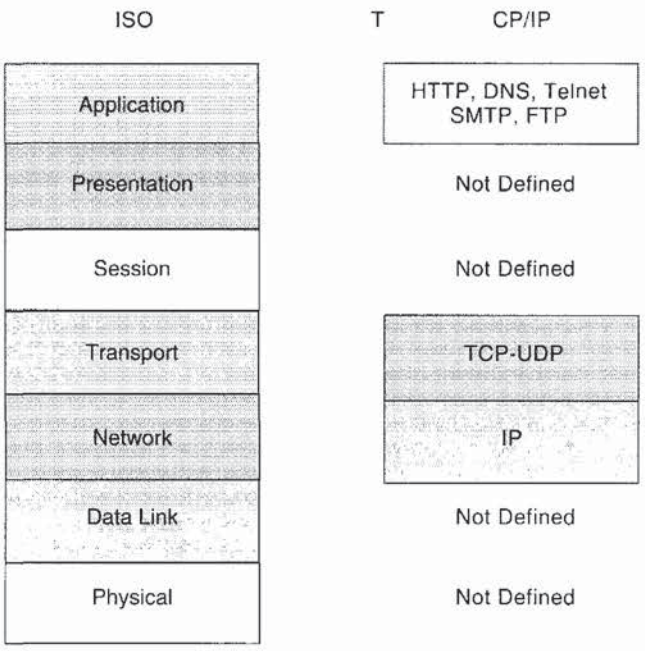


Figure 16.9 The ISO and TCP/IP protocol stacks.

To detect link and site failure, we use a **handshaking** procedure. Suppose that sites A and B have a direct physical link between them. At fixed intervals, the sites send each other an *I-am-up* message. If site A does not receive this message within a predetermined time period, it can assume that site B has failed, that the link between A and B has failed, or that the message from B has been lost. At this point, site A has two choices. It can wait for another time period to receive an *I-am-up* message from B, or it can send an *Are-you-up?* message to B.

If time goes by and site A still has not received an *I-am-up* message, or if site A has sent an *Are-you-up?* message and has not received a reply, the procedure can be repeated. Again, the only conclusion that site A can draw safely is that some type of failure has occurred.

Site A can try to differentiate between link failure and site failure by sending an *Are-you-up?* message to B by another route (if one exists). If and when B receives this message, it immediately replies positively. This positive reply tells A that B is up and that the failure is in the direct link between them. Since we do not know in advance how long it will take the message to travel from A to B and back, we must use a **time-out scheme**. At the time A sends the *Are-you-up?* message, it specifies a time interval during which it is willing to wait for the reply from B. If A receives the reply message within that time interval, then it can safely conclude that B is up. If not, however (that is, if a time-out occurs), then A may conclude only that one or more of the following situations has occurred:

- Site B is down.
- The direct link (if one exists) from A to B is down.

- The alternative path from A to B is down.
- The message has been lost.

Site A cannot, however, determine which of these events has occurred.

16.7.2 Reconfiguration

Suppose that site A has discovered, through the mechanism described in the previous section, that a failure has occurred. It must then initiate a procedure that will allow the system to reconfigure and to continue its normal mode of operation.

- If a direct link from A to B has failed, this information must be broadcast to every site in the system, so that the various routing tables can be updated accordingly.
- If the system believes that a site has failed (because that site can be reached no longer), then all sites in the system must be so notified, so that they will no longer attempt to use the services of the failed site. The failure of a site that serves as a central coordinator for some activity (such as deadlock detection) requires the election of a new coordinator. Similarly, if the failed site is part of a logical ring, then a new logical ring must be constructed. Note that, if the site has not failed (that is, if it is up but cannot be reached), then we may have the undesirable situation where two sites serve as the coordinator. When the network is partitioned, the two coordinators (each for its own partition) may initiate conflicting actions. For example, if the coordinators are responsible for implementing mutual exclusion, we may have a situation where two processes are executing simultaneously in their critical sections.

16.7.3 Recovery from Failure

When a failed link or site is repaired, it must be integrated into the system gracefully and smoothly.

- Suppose that a link between A and B has failed. When it is repaired, both A and B must be notified. We can accomplish this notification by continuously repeating the handshaking procedure described in Section 16.7.1.
- Suppose that site B has failed. When it recovers, it must notify all other sites that it is up again. Site B then may have to receive information from the other sites to update its local tables; for example, it may need routing-table information, a list of sites that are down, or undelivered messages and mail. If the site has not failed but simply could not be reached, then this information is still required.

16.8 Design Issues

Making the multiplicity of processors and storage devices **transparent** to the users has been a key challenge to many designers. Ideally, a distributed system

should look to its users like a conventional, centralized system. The user interface of a transparent distributed system should not distinguish between local and remote resources. That is, users should be able to access remote resources as though these resources were local, and the distributed system should be responsible for locating the resources and for arranging for the appropriate interaction.

Another aspect of transparency is user mobility. It would be convenient to allow users to log into any machine in the system rather than forcing them to use a specific machine. A transparent distributed system facilitates user mobility by bringing over the user's environment (for example, home directory) to wherever she logs in. Both the Andrew file system from CMU and Project Athena from MIT provide this functionality on a large scale; NFS can provide it on a smaller scale.

Another design issue involves fault tolerance. We use the term *fault tolerance* in a broad sense. Communication faults, machine failures (of type fail-stop), storage-device crashes, and decays of storage media should all be tolerated to some extent. A **fault-tolerant system** should continue to function, perhaps in a degraded form, when faced with these failures. The degradation can be in performance, in functionality, or in both. It should be proportional, however, to the failures that cause it. A system that grinds to a halt when only a few of its components fail is certainly not fault tolerant. Unfortunately, fault tolerance is difficult to implement. Most commercial systems provide only limited fault tolerance. For instance, the DEC VAX cluster allows multiple computers to share a set of disks. If a system crashes, users can still access their information from another system. Of course, if a disk fails, all systems will lose access. But in this case, RAID can ensure continued access to the data even in the event of a failure (Section 12.7).

Still another issue is **scalability**—the capability of a system to adapt to increased service load. Systems have bounded resources and can become completely saturated under increased load. For example, regarding a file system, saturation occurs either when a server's CPU runs at a high utilization rate or when disks are almost full. Scalability is a relative property, but it can be measured accurately. A scalable system reacts more gracefully to increased load than does a nonscalable one. First, its performance degrades more moderately; and second, its resources reach a saturated state later. Even perfect design cannot accommodate an ever-growing load. Adding new resources might solve the problem, but it might generate additional indirect load on other resources (for example, adding machines to a distributed system can clog the network and increase service loads). Even worse, expanding the system can call for expensive design modifications. A scalable system should have the potential to grow without these problems. In a distributed system, the ability to scale up gracefully is of special importance, since expanding the network by adding new machines or interconnecting two networks is commonplace. In short, a scalable design should withstand high service load, accommodate growth of the user community, and enable simple integration of added resources.

Fault tolerance and scalability are related to each other. A heavily loaded component can become paralyzed and behave like a faulty component. Also, shifting the load from a faulty component to that component's backup can saturate the latter. Generally, having spare resources is essential for ensuring reliability as well as for handling peak loads gracefully. An inherent advantage

of a distributed system is a potential for fault tolerance and scalability because of the multiplicity of resources. However, inappropriate design can obscure this potential. Fault-tolerance and scalability considerations call for a design demonstrating distribution of control and data.

Very large-scale distributed systems, to a great extent, are still only theoretical. No magic guidelines ensure the scalability of a system. It is easier to point out why current designs are *not* scalable. We next discuss several designs that pose problems and propose possible solutions, all in the context of scalability.

One principle for designing very large-scale systems is that the service demand from any component of the system should be bounded by a constant that is independent of the number of nodes in the system. Any service mechanism whose load demand is proportional to the size of the system is destined to become clogged once the system grows beyond a certain size. Adding more resources will not alleviate such a problem. The capacity of this mechanism simply limits the growth of the system.

Central control schemes and central resources should not be used to build scalable (and fault-tolerant) systems. Examples of centralized entities are central authentication servers, central naming servers, and central file servers. Centralization is a form of functional asymmetry among machines constituting the system. The ideal alternative is a functionally symmetric configuration; that is, all the component machines have an equal role in the operation of the system, and hence each machine has some degree of autonomy. Practically, it is virtually impossible to comply with such a principle. For instance, incorporating diskless machines violates functional symmetry, since the workstations depend on a central disk. However, autonomy and symmetry are important goals to which we should aspire.

The practical approximation of symmetric and autonomous configuration is **clustering**, in which the system is partitioned into a collection of semi-autonomous clusters. A **cluster** consists of a set of machines and a dedicated cluster server. So that cross-cluster resource references are relatively infrequent, each cluster server should satisfy requests of its own machines most of the time. Of course, this scheme depends on the ability to localize resource references and to place the component units appropriately. If the cluster is well balanced —that is, if the server in charge suffices to satisfy all the cluster demands—it can be used as a modular building block to scale up the system.

Deciding on the process structure of the server is a major problem in the design of any service. Servers are supposed to operate efficiently in peak periods, when hundreds of active clients need to be served simultaneously. A single-process server is certainly not a good choice, since whenever a request necessitates disk I/O, the whole service will be blocked. Assigning a process for each client is a better choice; however, the expense of frequent context switches between the processes must be considered. A related problem occurs because all the server processes need to share information.

One of the best solutions for the server architecture is the use of lightweight processes, or threads, which we discussed in Chapter 4. We can think of a group of lightweight processes as multiple threads of control associated with some shared resources. Usually, a lightweight process is not bound to a particular client. Instead, it serves single requests of different clients. Scheduling of threads can be preemptive or nonpreemptive. If threads are allowed to run

to completion (nonpreemptive), then their shared data do not need to be protected explicitly. Otherwise, some explicit locking mechanism must be used. Clearly, some form of lightweight-process scheme is essential if servers are to be scalable.

16.9 An Example: Networking

We now return to the name-resolution issue raised in Section 16.5.1 and examine its operation with respect to the TCP/IP protocol stack on the Internet. We consider the processing needed to transfer a packet between hosts on different Ethernet networks.

In a TCP/IP network, every host has a name and an associated 32-bit Internet number (or host-id). Both of these strings must be unique; and so that the name space can be managed, they are segmented. The name is hierarchical (as explained in Section 16.5.1), describing the host name and then the organization with which the host is associated. The host-id is split into a network number and a host number. The proportion of the split varies, depending on the size of the network. Once the Internet administrators assign a network number, the site with that number is free to assign host-ids.

The sending system checks its routing tables to locate a router to send the packet on its way. The routers use the network part of the host-id to transfer the packet from its source network to the destination network. The destination system then receives the packet. The packet may be a complete message, or it may just be a component of a message, with more packets needed before the message can be reassembled and passed to the TCP/UDP layer for transmission to the destination process.

Now we know how a packet moves from its source network to its destination. Within a network, how does a packet move from sender (host or router) to receiver? Every Ethernet device has a unique byte number, called the **medium access control (MAC) address**, assigned to it for addressing. Two devices on a LAN communicate with each other only with this number. If a system needs to send data to another system, the kernel generates an **address resolution protocol (ARP)** packet containing the IP address of the destination system. This packet is **broadcast** to all other systems on that Ethernet network.

A broadcast uses a special network address (usually, the maximum address) to signal that all hosts should receive and process the packet. The broadcast is not re-sent by gateways, so only systems on the local network receive it. Only the system whose IP address matches the IP address of the ARP request responds and sends back its MAC address to the system that initiated the query. For efficiency, the host caches the IP–MAC address pair in an internal table. The cache entries are **aged**, so that an entry is eventually removed from the cache if an access to that system is not required in a given time. In this way, hosts that are removed from a network are eventually *forgotten*. For added performance, ARP entries for heavily used hosts may be hardwired in the ARP cache.

Once an Ethernet device has announced its host-id and address, communication can begin. A process may specify the name of a host with which to communicate. The kernel takes that name and determines the Internet number of the target, using a DNS lookup. The message is passed from the application

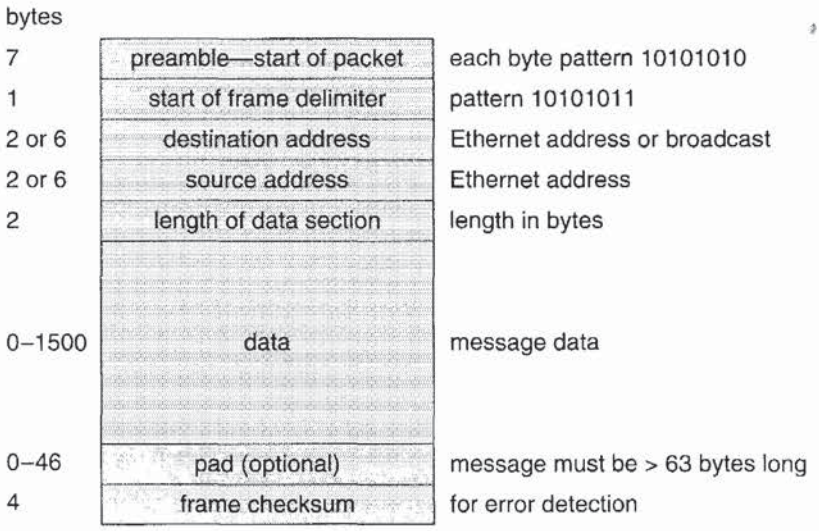


Figure 16.10 An Ethernet packet.

layer, through the software layers, and to the hardware layer. At the hardware layer, the packet (or packets) has the Ethernet address at its start; a trailer indicates the end of the packet and contains a **checksum** for detection of packet damage (Figure 16.10). The packet is placed on the network by the Ethernet device. The data section of the packet may contain some or all of the data of the original message, but it may also contain some of the upper-level headers that compose the message. In other words, all parts of the original message must be sent from source to destination, and all headers above the 802.3 layer (data-link layer) are included as data in the Ethernet packets.

If the destination is on the same local network as the source, the system can look in its ARP cache, find the Ethernet address of the host, and place the packet on the wire. The destination Ethernet device then sees its address in the packet and reads in the packet, passing it up the protocol stack.

If the destination system is on a network different from that of the source, the source system finds an appropriate router on its network and sends the packet there. Routers then pass the packet along the WAN until it reaches its destination network. The router that connects the destination network checks its ARP cache, finds the Ethernet number of the destination, and sends the packet to that host. Through all of these transfers, the data-link-layer header may change as the Ethernet address of the next router in the chain is used, but the other headers of the packet remain the same until the packet is received and processed by the protocol stack and finally passed to the receiving process by the kernel.

16.10 Summary

A distributed system is a collection of processors that do not share memory or a clock. Instead, each processor has its own local memory, and the processors communicate with one another through various communication lines, such

as high-speed buses and telephone lines. The processors in a distributed system vary in size and function. They may include small microprocessors, workstations, minicomputers, and large general-purpose computer systems.

The processors in the system are connected through a communication network, which can be configured in a number of ways. The network may be fully or partially connected. It may be a tree, a star, a ring, or a multiaccess bus. The communication-network design must include routing and connection strategies, and it must solve the problems of contention and security.

A distributed system provides the user with access to the resources the system provides. Access to a shared resource can be provided by data migration, computation migration, or process migration.

Protocol stacks, as specified by network layering models, massage the message, adding information to it to ensure that it reaches its destination. A naming system such as DNS must be used to translate from a host name to a network address, and another protocol (such as ARP) may be needed to translate the network number to a network device address (an Ethernet address, for instance). If systems are located on separate networks, routers are needed to pass packets from source network to destination network.

A distributed system may suffer from various types of hardware failure. For a distributed system to be fault tolerant, it must detect hardware failures and reconfigure the system. When the failure is repaired, the system must be reconfigured again.

Exercises

16.1 What is the difference between computation migration and process migration? Which is easier to implement, and why?

16.2 Contrast the various network topologies in terms of the following attributes:

a. Reliability

b. Available bandwidth for concurrent communications

c. Installation cost

d. Load balance in routing responsibilities

16.3 Even though the ISO model of networking specifies seven layers of functionality, most computer systems use fewer layers to implement a network. Why do they use fewer layers? What problems could the use of fewer layers cause?

16.4 Explain why doubling the speed of the systems on an Ethernet segment may result in decreased network performance. What changes could help solve this problem?

16.5 What are the advantages of using dedicated hardware devices for routers and gateways? What are the disadvantages of using these devices compared with using general-purpose computers?

16.6 In what ways is using a name server better than using static host tables? What problems or complications are associated with name servers? What methods could you use to decrease the amount of traffic name servers generate to satisfy translation requests?

16.7 Name servers are organized in a hierarchical manner. What is the purpose of using a hierarchical organization?

16.8 Consider a network layer that senses collisions and retransmits immediately on detection of a collision. What problems could arise with this strategy? How could they be rectified?

16.9 The lower layers of the ISO network model provide datagram service, with no delivery guarantees for messages. A transport-layer protocol such as TCP is used to provide reliability. Discuss the advantages and disadvantages of supporting reliable message delivery at the lowest possible layer.

16.10 What are the implications of using a dynamic routing strategy on application behavior? For what type of applications is it beneficial to use virtual routing instead of dynamic routing?

16.11 Run the program shown in Figure 16.5 and determine the IP addresses of the following host names:

- www.wiley.com
- www.cs.yale.edu
- www.javasoft.com
- www.westminstercollege.edu
- www.ietf.org

16.12 Consider a distributed system with two sites, A and B. Consider whether site A can distinguish among the following:

a. B goes down.

b. The link between A and B goes down.

c. B is extremely overloaded and its response time is 100 times longer than normal.

What implications does your answer have for recovery in distributed systems?

16.13 The original HTTP protocol used TCP/IP as the underlying network protocol. For each page, graphic, or applet, a separate TCP session was constructed, used, and torn down. Because of the overhead of building and destroying TCP/IP connections, performance problems resulted from this implementation method. Would using UDP rather than TCP be a good alternative? What other changes could you make to improve HTTP performance?

16.14 Of what use is an address-resolution protocol? Why is it better to use such a protocol than to make each host read each packet to determine

that packet's destination? Does a token-passing network need such a protocol? Explain your answer.

16.15 What are the advantages and the disadvantages of making the computer network transparent to the user?

Bibliographical Notes

Tanenbaum [2003], Stallings [2000a], and Kurose and Ross [2005] provided general overviews of computer networks. Williams [2001] covered computer networking from a computer-architecture viewpoint.

The Internet and its protocols were described in Comer [1999] and Comer [2000]. Coverage of TCP/IP can be found in Stevens [1994] and Stevens [1995]. UNIX network programming was described thoroughly in Stevens [1997] and Stevens [1998].

Discussions concerning distributed operating-system structures have been offered by Coulouris et al. [2001] and Tanenbaum and van Steen [2002].

Load balancing and load sharing were discussed by Harchol-Balter and Downey [1997] and Vee and Hsu [2000]. Harish and Owens [1999] described load-balancing DNS servers. Process migration was discussed by Jul et al. [1988], Douglis and Ousterhout [1991], Han and Ghosh [1998] and Milojicic et al. [2000]. Issues relating to a distributed virtual machine for distributed systems were examined in Sirer et al. [1999].

Distributed File Systems



In the previous chapter, we discussed network construction and the low-level protocols needed for messages to be transferred between systems. Now we examine one use of this infrastructure. A **distributed file system (DFS)** is a distributed implementation of the classical time-sharing model of a file system, where multiple users share files and storage resources (Chapter 11). The purpose of a DFS is to support the same kind of sharing when the files are physically dispersed among the sites of a distributed system.

In this chapter, we describe how a DFS can be designed and implemented. First, we discuss common concepts on which DFSs are based. Then, we illustrate our concepts by examining one influential DFS—the Andrew file system (AFS).

CHAPTER OBJECTIVES

- To explain the naming mechanism that provides location transparency and independence.
- To describe the various methods for accessing distributed files.
- To contrast stateful and stateless distributed file servers.
- To show how replication of files on different machines in a is a useful redundancy for improving availability. file replication
- To introduce the Andrew file system (AFS) as an example of a distributed file system.

17.1 Background

As we noted in the preceding chapter, a distributed system is a collection of loosely coupled computers interconnected by a communication network. These computers can share physically dispersed files by using a distributed file system (DFS). In this chapter, we use the term *DFS* to mean distributed file systems in general, not the commercial Transarc DFS product. The latter is referenced as *Transarc DFS*. Also, NFS refers to NFS version 3, unless otherwise noted.

To explain the structure of a DFS, we need to define the terms *service*, *server*, and *client*. A **service** is a software entity running on one or more machines and providing a particular type of function to clients. A **server** is the service software running on a single machine. A **client** is a process that can invoke a service using a set of operations that form its **client interface**. Sometimes a lower-level interface is defined for the actual cross-machine interaction; it is the **intermachine interface**.

Using this terminology, we say that a file system provides file services to clients. A client interface for a file service is formed by a set of primitive file operations, such as create a file, delete a file, read from a file, and write to a file. The primary hardware component that a file server controls is a set of local secondary-storage devices (usually, magnetic disks) on which files are stored and from which they are retrieved according to the clients' requests.

A DFS is a file system whose clients, servers, and storage devices are dispersed among the machines of a distributed system. Accordingly, service activity has to be carried out across the network. Instead of a single centralized data repository, the system frequently has multiple and independent storage devices. As you will see in this text, the concrete configuration and implementation of a DFS may vary from system to system. In some configurations, servers run on dedicated machines; in others, a machine can be both a server and a client. A DFS can be implemented as part of a distributed operating system or, alternatively, by a software layer whose task is to manage the communication between conventional operating systems and file systems. The distinctive features of a DFS are the multiplicity and autonomy of clients and servers in the system.

Ideally, a DFS should appear to its clients to be a conventional, centralized file system. The multiplicity and dispersion of its servers and storage devices should be made invisible. That is, the client interface of a DFS should not distinguish between local and remote files. It is up to the DFS to locate the files and to arrange for the transport of the data. A **transparent** DFS facilitates user mobility by bringing the user's environment (that is, home directory) to wherever a user logs in.

The most important performance measurement of a DFS is the amount of time needed to satisfy service requests. In conventional systems, this time consists of disk-access time and a small amount of CPU-processing time. In a DFS, however, a remote access has the additional overhead attributed to the distributed structure. This overhead includes the time to deliver the request to a server, as well as the time to get the response across the network back to the client. For each direction, in addition to the transfer of the information, there is the CPU overhead of running the communication protocol software. The performance of a DFS can be viewed as another dimension of the DFS's transparency. That is, the performance of an ideal DFS would be comparable to that of a conventional file system.

The fact that a DFS manages a set of dispersed storage devices is the DFS's key distinguishing feature. The overall storage space managed by a DFS is composed of different and remotely located smaller storage spaces. Usually, these constituent storage spaces correspond to sets of files. A **component unit** is the smallest set of files that can be stored on a single machine, independently from other units. All files belonging to the same component unit must reside in the same location.

17.2 Naming and Transparency

Naming is a mapping between logical and physical objects. For instance, users deal with logical data objects represented by file names, whereas the system manipulates physical blocks of data stored on disk tracks. Usually, a user refers to a file by a textual name. The latter is mapped to a lower-level numerical identifier that in turn is mapped to disk blocks. This multilevel mapping provides users with an abstraction of a file that hides the details of how and where on the disk the file is stored.

In a transparent DFS, a new dimension is added to the abstraction: that of hiding where in the network the file is located. In a conventional file system, the range of the naming mapping is an address within a disk. In a DFS, this range is expanded to include the specific machine on whose disk the file is stored. Going one step further with the concept of treating files as abstractions leads to the possibility of **file replication**. Given a file name, the mapping returns a set of the locations of this file's replicas. In this abstraction, both the existence of multiple copies and their locations are hidden.

17.2.1 Naming Structures

We need to differentiate two related notions regarding name mappings in a DFS:

1. **Location transparency**. The name of a file does not reveal any hint of the file's physical storage location.
2. **Location independence**. The name of a file does not need to be changed when the file's physical storage location changes.

Both definitions are relative to the level of naming discussed previously, since files have different names at different levels (that is, user-level textual names and system-level numerical identifiers). A location-independent naming scheme is a dynamic mapping, since it can map the same file name to different locations at two different times. Therefore, location independence is a stronger property than is location transparency.

In practice, most of the current DFSs provide a static, location-transparent mapping for user-level names. These systems, however, do not support **file migration**; that is, changing the location of a file automatically is impossible. Hence, the notion of location independence is irrelevant for these systems. Files are associated permanently with a specific set of disk blocks. Files and disks can be moved between machines manually, but file migration implies an automatic, operating-system-initiated action. Only AFS and a few experimental file systems support location independence and file mobility. AFS supports file mobility mainly for administrative purposes. A protocol provides migration of AFS component units to satisfy high-level user requests, without changing either the user-level names or the low-level names of the corresponding files.

A few aspects can further differentiate location independence and static location transparency:

- Divorce of data from location, as exhibited by location independence, provides a better abstraction for files. A file name should denote the file's

most significant attributes, which are its contents rather than its location. Location-independent files can be viewed as logical data containers that are not attached to a specific storage location. If only static location transparency is supported, the file name still denotes a specific, although hidden, set of physical disk blocks.

- Static location transparency provides users with a convenient way to share data. Users can share remote files by simply naming the files in a location-transparent manner, as though the files were local. Nevertheless, sharing the storage space is cumbersome, because logical names are still statically attached to physical storage devices. Location independence promotes sharing the storage space itself, as well as the data objects. When files can be mobilized, the overall, system-wide storage space looks like a single virtual resource. A possible benefit of such a view is the ability to balance the utilization of disks across the system.
- Location independence separates the naming hierarchy from the storage-devices hierarchy and from the intercomputer structure. By contrast, if static location transparency is used (although names are transparent), we can easily expose the correspondence between component units and machines. The machines are configured in a pattern similar to the naming structure. This configuration may restrict the architecture of the system unnecessarily and conflict with other considerations. A server in charge of a root directory is an example of a structure that is dictated by the naming hierarchy and contradicts decentralization guidelines.

Once the separation of name and location has been completed, clients can access files residing on remote server systems. In fact, these clients may be **diskless** and rely on servers to provide all files, including the operating-system kernel. Special protocols are needed for the boot sequence, however. Consider the problem of getting the kernel to a diskless workstation. The diskless workstation has no kernel, so it cannot use the DFS code to retrieve the kernel. Instead, a special boot protocol, stored in read-only memory (ROM) on the client, is invoked. It enables networking and retrieves only one special file (the kernel or boot code) from a fixed location. Once the kernel is copied over the network and loaded, its DFS makes all the other operating-system files available. The advantages of diskless clients are many, including lower cost (because the client machines require no disks) and greater convenience (when an operating-system upgrade occurs, only the server needs to be modified). The disadvantages are the added complexity of the boot protocols and the performance loss resulting from the use of a network rather than a local disk.

The current trend is for clients to use both local disks and remote file servers. Operating systems and networking software are stored locally; file systems containing user data—and possibly applications—are stored on remote file systems. Some client systems may store commonly used applications, such as word processors and web browsers, on the local file system as well. Other, less commonly used applications may be **pushed** from the remote file server to the client on demand. The main reason for providing clients with local file systems rather than pure diskless systems is that disk drives are rapidly increasing in capacity and decreasing in cost, with new generations appearing every year or so. The same cannot be said for networks, which evolve every few years.

Overall, systems are growing more quickly than are networks, so extra work is needed to limit network access to improve system throughput.

17.2.2 Naming Schemes

There are three main approaches to naming schemes in a DFS. In the simplest approach, a file is identified by some combination of its host name and local name, which guarantees a unique system-wide name. In Ibis, for instance, a file is identified uniquely by the name *host:local-name*, where *local-name* is a UNIX-like path. This naming scheme is neither location transparent nor location independent. Nevertheless, the same file operations can be used for both local and remote files. The DFS is structured as a collection of isolated component units, each of which is an entire conventional file system. In this first approach, component units remain isolated, although means are provided to refer to a remote file. We do not consider this scheme any further in this text.

The second approach was popularized by Sun's network file system (NFS). NFS is the file-system component of ONC+, a networking package supported by many UNIX vendors. NFS provides a means to attach remote directories to local directories, thus giving the appearance of a coherent directory tree. Early NFS versions allowed only previously mounted remote directories to be accessed transparently. With the advent of the **automount** feature, mounts are done on demand, based on a table of mount points and file-structure names. Components are integrated to support transparent sharing, although this integration is limited and is not uniform, because each machine may attach different remote directories to its tree. The resulting structure is versatile.

We can achieve total integration of the component file systems by using the third approach. A single global name structure spans all the files in the system. Ideally, the composed file-system structure is isomorphic to the structure of a conventional file system. In practice, however, the many special files (for example, UNIX device files and machine-specific binary directories) make this goal difficult to attain.

To evaluate naming structures, we look at their **administrative complexity**. The most complex and most difficult-to-maintain structure is the NFS structure. Because any remote directory can be attached anywhere onto the local directory tree, the resulting hierarchy can be highly unstructured. If a server becomes unavailable, some arbitrary set of directories on different machines becomes unavailable. In addition, a separate accreditation mechanism controls which machine is allowed to attach which directory to its tree. Thus, a user might be able to access a remote directory tree on one client but be denied access on another client.

17.2.3 Implementation Techniques

Implementation of transparent naming requires a provision for the mapping of a file name to the associated location. To keep this mapping manageable, we must aggregate sets of files into component units and provide the mapping on a component-unit basis rather than on a single-file basis. This aggregation serves administrative purposes as well. UNIX-like systems use the hierarchical directory tree to provide name-to-location mapping and to aggregate files recursively into directories.

To enhance the availability of the crucial mapping information, we can use replication, local caching, or both. As we noted, location independence means that the mapping changes over time; hence, replicating the mapping makes a simple yet consistent update of this information impossible. A technique to overcome this obstacle is to introduce low-level **location-independent file identifiers**. Textual file names are mapped to lower-level file identifiers that indicate to which component unit the file belongs. These identifiers are still location independent. They can be replicated and cached freely without being invalidated by migration of component units. The inevitable price is the need for a second level of mapping, which maps component units to locations and needs a simple yet consistent update mechanism. Implementing UNIX-like directory trees using these low-level, location-independent identifiers makes the whole hierarchy invariant under component-unit migration. The only aspect that does change is the component-unit location mapping.

A common way to implement low-level identifiers is to use structured names. These names are bit strings that usually have two parts. The first part identifies the component unit to which the file belongs; the second part identifies the particular file within the unit. Variants with more parts are possible. The invariant of structured names, however, is that individual parts of the name are unique at all times only within the context of the rest of the parts. We can obtain uniqueness at all times by taking care not to reuse a name that is still used, by adding sufficiently more bits (this method is used in AFS), or by using a timestamp as one part of the name (as done in Apollo Domain). Another way to view this process is that we are taking a location-transparent system, such as Ibis, and adding another level of abstraction to produce a location-independent naming scheme.

Aggregating files into component units and using lower-level location-independent file identifiers are techniques exemplified in AFS.

17.3 Remote File Access

Consider a user who requests access to a remote file. The server storing the file has been located by the naming scheme, and now the actual data transfer must take place.

One way to achieve this transfer is through a **remote-service mechanism**, whereby requests for accesses are delivered to the server, the server machine performs the accesses, and their results are forwarded back to the user. One of the most common ways of implementing remote service is the remote procedure call (RPC) paradigm, which we discussed in Chapter 3. A direct analogy exists between disk-access methods in conventional file systems and the remote-service method in a DFS: Using the remote-service method is analogous to performing a disk access for each access request.

To ensure reasonable performance of a remote-service mechanism, we can use a form of caching. In conventional file systems, the rationale for caching is to reduce disk I/O (thereby increasing performance), whereas in DFSs, the goal is to reduce both network traffic and disk I/O. In the following discussion, we describe the implementation of caching in a DFS and contrast it with the basic remote-service paradigm.

17.3.1 Basic Caching Scheme

The concept of caching is simple. If the data needed to satisfy the access request are not already cached, then a copy of those data is brought from the server to the client system. Accesses are performed on the cached copy. The idea is to retain recently accessed disk blocks in the cache, so that repeated accesses to the same information can be handled locally, without additional network traffic. A replacement policy (for example, least recently used) keeps the cache size bounded. No direct correspondence exists between accesses and traffic to the server. Files are still identified with one master copy residing at the server machine, but copies (or parts) of the file are scattered in different caches. When a cached copy is modified, the changes need to be reflected on the master copy to preserve the relevant consistency semantics. The problem of keeping the cached copies consistent with the master file is the **cache-consistency problem**, which we discuss in Section 17.3.4. DFS caching could just as easily be called **network virtual memory**; it acts similarly to demand-paged virtual memory, except that the backing store usually is not a local disk but rather a remote server. NFS allows the swap space to be mounted remotely, so it actually can implement virtual memory over a network, notwithstanding the resulting performance penalty.

The granularity of the cached data in a DFS can vary from blocks of a file to an entire file. Usually, more data are cached than are needed to satisfy a single access, so that many accesses can be served by the cached data. This procedure is much like disk read-ahead (Section 11.6.2). AFS caches files in large chunks (64 KB). The other systems discussed in this chapter support caching of individual blocks driven by client demand. Increasing the caching unit increases the hit ratio, but it also increases the miss penalty, because each miss requires more data to be transferred. It increases the potential for consistency problems as well. Selecting the unit of caching involves considering parameters such as the network transfer unit and the RPC protocol service unit (if an RPC protocol is used). The network transfer unit (for Ethernet, a packet) is about 1.5 KB, so larger units of cached data need to be disassembled for delivery and reassembled on reception.

Block size and total cache size are obviously of importance for block-caching schemes. In UNIX-like systems, common block sizes are 4 KB and 8 KB. For large caches (over 1 MB), large block sizes (over 8 KB) are beneficial. For smaller caches, large block sizes are less beneficial because they result in fewer blocks in the cache and a lower hit ratio.

17.3.2 Cache Location

Where should the cached data be stored—on disk or in main memory? Disk caches have one clear advantage over main-memory caches: They are reliable. Modifications to cached data are lost in a crash if the cache is kept in volatile memory. Moreover, if the cached data are kept on disk, they are still there during recovery, and there is no need to fetch them again. Main-memory caches have several advantages of their own, however:

- Main-memory caches permit workstations to be diskless.
- Data can be accessed more quickly from a cache in main memory than from one on a disk.

- Technology is moving toward larger and less expensive memory. The achieved performance speedup is predicted to outweigh the advantages of disk caches.
- The server caches (used to speed up disk I/O) will be in main memory regardless of where user caches are located; if we use main-memory caches on the user machine, too, we can build a single caching mechanism for use by both servers and users.

Many remote-access implementations can be thought of as hybrids of caching and remote service. In NFS, for instance, the implementation is based on remote service but is augmented with client- and server-side memory caching for performance. Similarly, Sprite's implementation is based on caching; but under certain circumstances, a remote-service method is adopted. Thus, to evaluate the two methods, we must evaluate to what degree either method is emphasized.

The NFS protocol and most implementations do not provide disk caching. Recent Solaris implementations of NFS (Solaris 2.6 and beyond) include a client-side disk caching option, the **cachefs** file system. Once the NFS client reads blocks of a file from the server, it caches them in memory as well as on disk. If the memory copy is flushed, or even if the system reboots, the disk cache is referenced. If a needed block is neither in memory nor in the cachefs disk cache, an RPC is sent to the server to retrieve the block, and the block is written into the disk cache as well as stored in the memory cache for client use.

17.3.3 Cache-Update Policy

The policy used to write modified data blocks back to the server's master copy has a critical effect on the system's performance and reliability. The simplest policy is to write data through to disk as soon as they are placed in any cache. The advantage of a **write-through policy** is reliability: Little information is lost when a client system crashes. However, this policy requires each write access to wait until the information is sent to the server, so it causes poor write performance. Caching with write-through is equivalent to using remote service for write accesses and exploiting caching only for read accesses.

An alternative is the **delayed-write policy**, also known as **write-back caching**, where we delay updates to the master copy. Modifications are written to the cache and then are written through to the server at a later time. This policy has two advantages over write-through. First, because writes are made to the cache, write accesses complete much more quickly. Second, data may be overwritten before they are written back, in which case only the last update needs to be written at all. Unfortunately, delayed-write schemes introduce reliability problems, since unwritten data are lost whenever a user machine crashes.

Variations of the delayed-write policy differ in when modified data blocks are flushed to the server. One alternative is to flush a block when it is about to be ejected from the client's cache. This option can result in good performance, but some blocks can reside in the client's cache a long time before they are written back to the server. A compromise between this alternative and the write-through policy is to scan the cache at regular intervals and to flush blocks that have been modified since the most recent scan, just as UNIX scans

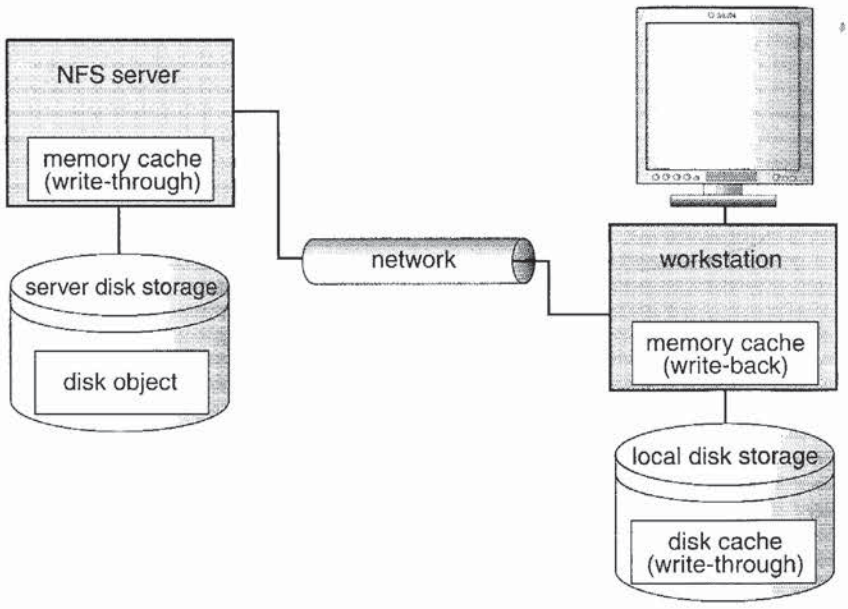


Figure 17.1 Cachefs and its use of caching.

its local cache. Sprite uses this policy with a 30-second interval. NFS uses the policy for file data, but once a write is issued to the server during a cache flush, the write must reach the server's disk before it is considered complete. NFS treats metadata (directory data and file-attribute data) differently. Any metadata changes are issued synchronously to the server. Thus, file-structure loss and directory-structure corruption are avoided when a client or the server crashes.

For NFS with cachefs, writes are also written to the local disk cache area when they are written to the server, to keep all copies consistent. Thus, NFS with cachefs improves performance over standard NFS on a read request with a cachefs cache hit but decreases performance for read or write requests with a cache miss. As with all caches, it is vital to have a high cache hit rate to gain performance. Cachefs and its use of write-through and write-back caching is shown in Figure 17.1.

Yet another variation on delayed write is to write data back to the server when the file is closed. This **write-on-close policy** is used in AFS. In the case of files that are open for short periods or are modified rarely, this policy does not significantly reduce network traffic. In addition, the write-on-close policy requires the closing process to delay while the file is written through, which reduces the performance advantages of delayed writes. For files that are open for long periods and are modified frequently, however, the performance advantages of this policy over delayed write with more frequent flushing are apparent.

17.3.4 Consistency

A client machine is faced with the problem of deciding whether or not a locally cached copy of the data is consistent with the master copy (and hence can be

used). If the client machine determines that its cached data are out of date, accesses can no longer be served by those cached data. An up-to-date copy of the data needs to be cached. There are two approaches to verifying the validity of cached data:

1. **Client-initiated approach.** The client initiates a validity check in which it contacts the server and checks whether the local data are consistent with the master copy. The frequency of the validity checking is the crux of this approach and determines the resulting consistency semantics. It can range from a check before every access to a check only on first access to a file (on file open, basically). Every access coupled with a validity check is delayed, compared with an access served immediately by the cache. Alternatively, checks can be initiated at fixed time intervals. Depending on its frequency, the validity check can load both the network and the server.
2. **Server-initiated approach.** The server records, for each client, the files (or parts of files) that it caches. When the server detects a potential inconsistency, it must react. A potential for inconsistency occurs when two different clients in conflicting modes cache a file. If UNIX semantics (Section 10.5.3) is implemented, we can resolve the potential inconsistency by having the server play an active role. The server must be notified whenever a file is opened, and the intended mode (read or write) must be indicated for every open. The server can then act when it detects that file has been opened simultaneously in conflicting modes by disabling caching for that particular file. Actually, disabling caching results in switching to a remote-service mode of operation.

17.3.5 A Comparison of Caching and Remote Service

Essentially, the choice between caching and remote service trades off potentially increased performance with decreased simplicity. We evaluate this tradeoff by listing the advantages and disadvantages of the two methods:

- When caching is used, the local cache can handle a substantial number of the remote accesses efficiently. Capitalizing on locality in file-access patterns makes caching even more attractive. Thus, most of the remote accesses will be served as fast as will local ones. Moreover, servers are contacted only occasionally, rather than for each access. Consequently, server load and network traffic are reduced, and the potential for scalability is enhanced. By contrast, when the remote-service method is used, every remote access is handled across the network. The penalty in network traffic, server load, and performance is obvious.
- Total network overhead is lower for transmitting big chunks of data (as is done in caching) than for transmitting series of responses to specific requests (as in the remote-service method). Furthermore, disk-access routines on the server may be better optimized if it is known that requests will always be for large, contiguous segments of data rather than for random disk blocks.

- The cache-consistency problem is the major drawback of caching. When access patterns exhibit infrequent writes, caching is superior. However, when writes are frequent, the mechanisms employed to overcome the consistency problem incur substantial overhead in terms of performance, network traffic, and server load.
- So that caching will confer a benefit, execution should be carried out on machines that have either local disks or large main memories. Remote access on diskless, small-memory-capacity machines should be done through the remote-service method.
- In caching, since data are transferred en masse between the server and the client, rather than in response to the specific needs of a file operation, the lower-level intermachine interface is different from the upper-level user interface. The remote-service paradigm, in contrast, is just an extension of the local file-system interface across the network. Thus, the intermachine interface mirrors the user interface.

17.4 Stateful Versus Stateless Service

There are two approaches for storing server-side information when a client accesses remote files: Either the server tracks each file being accessed by each client, or it simply provides blocks as they are requested by the client without knowledge of how those blocks are used. In the former case, the service provided is *stateful;* in the latter case, it is *stateless.*

The typical scenario of a **stateful file service** is as follows: A client must perform an `open()` operation on a file before accessing that file. The server fetches information about the file from its disk, stores it in its memory, and gives the client a connection identifier that is unique to the client and the open file. (In UNIX terms, the server fetches the inode and gives the client a file descriptor, which serves as an index to an in-core table of inodes.) This identifier is used for subsequent accesses until the session ends. A stateful service is characterized as a connection between the client and the server during a session. Either on closing the file or by a garbage-collection mechanism, the server must reclaim the main-memory space used by clients that are no longer active. The key point regarding fault tolerance in a stateful service approach is that the server keeps main-memory information about its clients. AFS is a stateful file service.

A **stateless file service** avoids state information by making each request self-contained. That is, each request identifies the file and the position in the file (for read and write accesses) in full. The server does not need to keep a table of open files in main memory, although it usually does so for efficiency reasons. Moreover, there is no need to establish and terminate a connection through `open()` and `close()` operations. They are totally redundant, since each file operation stands on its own and is not considered part of a session. A client process would open a file, and that open would not result in the sending of a remote message. Reads and writes would take place as remote messages (or cache lookups). The final close by the client would again result in only a local operation. NFS is a stateless file service.

The advantage of a stateful over a stateless service is increased performance. File information is cached in main memory and can be accessed easily

via the connection identifier, thereby saving disk accesses. In addition, a stateful server knows whether a file is open for sequential access and can therefore read ahead the next blocks. Stateless servers cannot do so, since they have no knowledge of the purpose of the client's requests.

The distinction between stateful and stateless service becomes more evident when we consider the effects of a crash that occurs during a service activity. A stateful server loses all its volatile state in a crash. Ensuring the graceful recovery of such a server involves restoring this state, usually by a recovery protocol based on a dialog with clients. Less graceful recovery requires that the operations that were underway when the crash occurred be aborted. A different problem is caused by client failures. The server needs to become aware of such failures so that it can reclaim space allocated to record the state of crashed client processes. This phenomenon is sometimes referred to as **orphan detection and elimination**.

A stateless computer server avoids these problems, since a newly reincarnated server can respond to a self-contained request without any difficulty. Therefore, the effects of server failures and recovery are almost unnoticeable. There is no difference between a slow server and a recovering server from a client's point of view. The client keeps retransmitting its request if it receives no response.

The penalty for using the robust stateless service is longer request messages and slower processing of requests, since there is no in-core information to speed the processing. In addition, stateless service imposes additional constraints on the design of the DFS. First, since each request identifies the target file, a uniform, system-wide, low-level naming scheme should be used. Translating remote to local names for each request would cause even slower processing of the requests. Second, since clients retransmit requests for file operations, these operations must be idempotent; that is, each operation must have the same effect and return the same output if executed several times consecutively. Self-contained read and write accesses are idempotent, as long as they use an absolute byte count to indicate the position within the file they access and do not rely on an incremental offset (as done in UNIX `read()` and `write()` system calls). However, we must be careful when implementing destructive operations (such as deleting a file) to make them idempotent, too.

In some environments, a stateful service is a necessity. If the server employs the server-initiated method for cache validation, it cannot provide stateless service, since it maintains a record of which files are cached by which clients.

The way UNIX uses file descriptors and implicit offsets is inherently stateful. Servers must maintain tables to map the file descriptors to inodes and must store the current offset within a file. This requirement is why NFS, which employs a stateless service, does not use file descriptors and does include an explicit offset in every access.

17.5 File Replication

Replication of files on different machines in a distributed file system is a useful redundancy for improving availability. Multimachine replication can benefit performance too: Selecting a nearby replica to serve an access request results in shorter service time.

NFS V4

Our coverage of NFS thus far has only considered version 3 (or V3) NFS. The most recent NFS standard is version 4 (V4), and it differs fundamentally from previous versions. The most significant change is that the protocol is now *stateful*, meaning that the server maintains the state of the client session from the time the remote file is opened until it is closed. Thus, the NFS protocol now provides `open()` and `close()` operations; previous versions of NFS (which are stateless) provide no such operations. Furthermore, previous versions specify separate protocols for mounting remote file systems and for locking remote files. V4 provides all of these features under a single protocol. In particular, the *mount* protocol was eliminated, allowing NFS to work with network firewalls. The *mount* protocol was a notorious security hole in NFS implementations.

Additionally, V4 has enhanced the ability of clients to cache file data locally. This feature improves the performance of the distributed file system, as clients are able to resolve more file accesses from the local cache rather than having to go through the server. V4 allows clients to request file locks from servers as well. If the server grants the request, the client maintains the lock until it is released or its lease expires. (Clients are also permitted to renew existing leases.) Traditionally, UNIX-based systems provide advisory file locking, whereas Windows operating systems use mandatory locking. To allow NFS to work well with non-UNIX systems, V4 now provides mandatory locking as well. The new locking and caching mechanisms are based on the concept of **delegation**, whereby the server delegates responsibilities for a file's lock and contents to the client that requested the lock. That delegated client maintains in cache the current version of the file, and other clients can ask that delegated client for lock access and file contents until the delegated client relinquishes the lock and delegation.

Finally, whereas previous versions of NFS are based on the UDP network protocol, V4 is based on TCP, which allows it to better adjust to varying traffic loads on the network. Delegating these responsibilities to clients reduces the load on the server and improves cache coherency.

The basic requirement of a replication scheme is that different replicas of the same file reside on failure-independent machines. That is, the availability of one replica is not affected by the availability of the rest of the replicas. This obvious requirement implies that replication management is inherently a location-opaque activity. Provisions for placing a replica on a particular machine must be available.

It is desirable to hide the details of replication from users. Mapping a replicated file name to a particular replica is the task of the naming scheme. The existence of replicas should be invisible to higher levels. At lower levels, however, the replicas must be distinguished from one another by different lower-level names. Another transparency requirement is providing replication control at higher levels. Replication control includes determination of the degree of replication and of the placement of replicas. Under certain circumstances, we may want to expose these details to users. Locus, for

instance, provides users and system administrators with mechanisms to control the replication scheme.

The main problem associated with replicas is updating. From a user's point of view, replicas of a file denote the same logical entity, and thus an update to any replica must be reflected on all other replicas. More precisely, the relevant consistency semantics must be preserved when accesses to replicas are viewed as virtual accesses to the replicas' logical files. If consistency is not of primary importance, it can be sacrificed for availability and performance. In this fundamental tradeoff in the area of fault tolerance, the choice is between preserving consistency at all costs, thereby creating a potential for indefinite blocking, and sacrificing consistency under some (we hope, rare) circumstances of catastrophic failures for the sake of guaranteed progress. Locus, for example, employs replication extensively and sacrifices consistency in the case of network partition for the sake of availability of files for read and write accesses.

Ibis uses a variation of the primary-copy approach. The domain of the name mapping is a pair *<primary-replica-identifier, local-replica-identifier>*. If no local replica exists, a special value is used. Thus, the mapping is relative to a machine. If the local replica is the primary one, the pair contains two identical identifiers. Ibis supports demand replication, an automatic replication-control policy similar to whole-file caching. Under demand replication, reading of a nonlocal replica causes it to be cached locally, thereby generating a new nonprimary replica. Updates are performed only on the primary copy and cause all other replicas to be invalidated through the sending of appropriate messages. Atomic and serialized invalidation of all nonprimary replicas is not guaranteed. Hence, a stale replica may be considered valid. To satisfy remote write accesses, we migrate the primary copy to the requesting machine.

17.6 An Example: AFS

Andrew is a distributed computing environment designed and implemented at Carnegie Mellon University. The Andrew file system (AFS) constitutes the underlying information-sharing mechanism among clients of the environment. The Transarc Corporation took over development of AFS, then was purchased by IBM. IBM has since produced several commercial implementations of AFS. AFS was subsequently chosen as the DFS for an industry coalition; the result was **Transarc DFS**, part of the distributed computing environment (DCE) from the OSF organization.

In 2000, IBM's Transarc Lab announced that AFS would be an open-source product (termed OpenAFS) available under the IBM public license and Transarc DFS was canceled as a commercial product. OpenAFS is available under most commercial versions of UNIX as well as Linux and Microsoft Windows systems. Many UNIX vendors, as well as Microsoft, support the DCE system and its DFS, which is based on AFS, and work is ongoing to make DCE a cross-platform, universally accepted DFS. As AFS and Transarc DFS are very similar, we describe AFS throughout this section, unless Transarc DFS is named specifically.

AFS seeks to solve many of the problems of the simpler DFSs, such as NFS, and is arguably the most feature-rich nonexperimental DFS. It features a uniform name space, location-independent file sharing, client-side caching

with cache consistency, and secure authentication via Kerberos. It also includes server-side caching in the form of replicas, with high availability through automatic switchover to a replica if the source server is unavailable. One of the most formidable attributes of AFS is scalability: The Andrew system is targeted to span over 5,000 workstations. Between AFS and Transarc DFS, there are hundreds of implementations worldwide.

17.6.1 Overview

AFS distinguishes between *client machines* (sometimes referred to as *workstations*) and dedicated *server machines*. Servers and clients originally ran only 4.2 BSD UNIX, but AFS has been ported to many operating systems. The clients and servers are interconnected by a network of LANs or WANs.

Clients are presented with a partitioned space of file names: a **local name space** and a **shared name space**. Dedicated servers, collectively called *Vice* after the name of the software they run, present the shared name space to the clients as a homogeneous, identical, and location-transparent file hierarchy. The local name space is the root file system of a workstation, from which the shared name space descends. Workstations run the *Virtue* protocol to communicate with Vice, and each is required to have a local disk where it stores its local name space. Servers collectively are responsible for the storage and management of the shared name space. The local name space is small, is distinct for each workstation, and contains system programs essential for autonomous operation and better performance. Also local are temporary files and files that the workstation owner, for privacy reasons, explicitly wants to store locally.

Viewed at a finer granularity, clients and servers are structured in clusters interconnected by a WAN. Each cluster consists of a collection of workstations on a LAN and a representative of Vice called a **cluster server**, and each cluster is connected to the WAN by a router. The decomposition into clusters is done primarily to address the problem of scale. For optimal performance, workstations should use the server on their own cluster most of the time, thereby making cross-cluster file references relatively infrequent.

The file-system architecture is also based on considerations of scale. The basic heuristic is to offload work from the servers to the clients, in light of experience indicating that server CPU speed is the system's bottleneck. Following this heuristic, the key mechanism selected for remote file operations is to cache files in large chunks (64 KB). This feature reduces file-open latency and allows reads and writes to be directed to the cached copy without frequently involving the servers.

Briefly, here are a few additional issues in the design of AFS:

- **Client mobility**. Clients are able to access any file in the shared name space from any workstation. A client may notice some initial performance degradation due to the caching of files when accessing files from a workstation other than the usual one.
- **Security**. The Vice interface is considered the boundary of trustworthiness, because no client programs are executed on Vice machines. Authentication and secure-transmission functions are provided as part of a connection-based communication package based on the RPC paradigm. After mutual

authentication, a Vice server and a client communicate via encrypted messages. Encryption is performed by hardware devices or (more slowly) in software. Information about clients and groups is stored in a protection database replicated at each server.

- **Protection**. AFS provides **access lists** for protecting directories and the regular UNIX bits for file protection. The access list may contain information about those users allowed to access a directory, as well as information about those users *not* allowed to access it. Thus, it is simple to specify that everyone except, say, Jim can access a directory. AFS supports the access types read, write, lookup, insert, administer, lock, and delete.
- **Heterogeneity**. Defining a clear interface to Vice is a key for integration of diverse workstation hardware and operating systems. So that heterogeneity is facilitated, some files in the local */bin* directory are symbolic links pointing to machine-specific executable files residing in Vice.

17.6.2 The Shared Name Space

AFS's shared name space is made up of component units called **volumes**. The volumes are unusually small component units. Typically, they are associated with the files of a single client. Few volumes reside within a single disk partition, and they may grow (up to a quota) and shrink in size. Conceptually, volumes are glued together by a mechanism similar to the UNIX mount mechanism. However, the granularity difference is significant, since in UNIX only an entire disk partition (containing a file system) can be mounted. Volumes are a key administrative unit and play a vital role in identifying and locating an individual file.

A Vice file or directory is identified by a low-level identifier called a **fid.** Each AFS directory entry maps a path-name component to a fid. A fid is 96 bits long and has three equal-length components: a *volume number*, a *vnode number*, and a *uniquifier*. The **vnode number** is used as an index into an array containing the inodes of files in a single volume. The **uniquifier** allows reuse of vnode numbers, thereby keeping certain data structures compact. Fids are location transparent; therefore, file movements from server to server do not invalidate cached directory contents.

Location information is kept on a volume basis in a **volume-location database** replicated on each server. A client can identify the location of every volume in the system by querying this database. The aggregation of files into volumes makes it possible to keep the location database at a manageable size.

To balance the available disk space and utilization of servers, volumes need to be migrated among disk partitions and servers. When a volume is shipped to its new location, its original server is left with temporary forwarding information, so that the location database does not need to be updated synchronously. While the volume is being transferred, the original server can still handle updates, which are shipped later to the new server. At some point, the volume is briefly disabled so that the recent modifications can be processed; then, the new volume becomes available again at the new site. The volume-movement operation is atomic; if either server crashes, the operation is aborted.

Read-only replication at the granularity of an entire volume is supported for system-executable files and for seldom-updated files in the upper levels of the Vice name space. The volume-location database specifies the server containing the only read–write copy of a volume and a list of read-only replication sites.

17.6.3 File Operations and Consistency Semantics

The fundamental architectural principle in AFS is the caching of entire files from servers. Accordingly, a client workstation interacts with Vice servers only during opening and closing of files, and even this interaction is not always necessary. Reading and writing files do not cause remote interaction (in contrast to the remote-service method). This key distinction has far-reaching ramifications for performance, as well as for semantics of file operations.

The operating system on each workstation intercepts file-system calls and forwards them to a client-level process on that workstation. This process, called *Venus*, caches files from Vice when they are opened and stores modified copies of files back on the servers from which they came when they are closed. Venus may contact Vice only when a file is opened or closed; reading and writing of individual bytes of a file are performed directly on the cached copy and bypass Venus. As a result, writes at some sites are not visible immediately at other sites.

Caching is further exploited for future opens of the cached file. Venus assumes that cached entries (files or directories) are valid unless notified otherwise. Therefore, Venus does not need to contact Vice on a file open to validate the cached copy. The mechanism to support this policy, called **callback**, dramatically reduces the number of cache-validation requests received by servers. It works as follows. When a client caches a file or a directory, the server updates its state information to record this caching. We say that the client has a callback on that file. The server notifies the client before allowing another client to modify the file. In such a case, we say that the server removes the callback on the file for the former client. A client can use a cached file for open purposes only when the file has a callback. If a client closes a file after modifying it, all other clients caching this file lose their callbacks. Therefore, when these clients open the file later, they have to get the new version from the server.

Reading and writing bytes of a file are done directly by the kernel without Venus's intervention on the cached copy. Venus regains control when the file is closed. If the file has been modified locally, it updates the file on the appropriate server. Thus, the only occasions on which Venus contacts Vice servers are on opens of files that either are not in the cache or have had their callback revoked and on closes of locally modified files.

Basically, AFS implements session semantics. The only exceptions are file operations other than the primitive read and write (such as protection changes at the directory level), which are visible everywhere on the network immediately after the operation completes.

In spite of the callback mechanism, a small amount of cached validation traffic is still present, usually to replace callbacks lost because of machine or network failures. When a workstation is rebooted, Venus considers all cached

files and directories suspect, and it generates a cache-validation request for the first use of each such entry.

The callback mechanism forces each server to maintain callback information and each client to maintain validity information. If the amount of callback information maintained by a server is excessive, the server can break callbacks and reclaim some storage by unilaterally notifying clients and revoking the validity of their cached files. If the callback state maintained by Venus gets out of sync with the corresponding state maintained by the servers, some inconsistency may result.

Venus also caches contents of directories and symbolic links, for path-name translation. Each component in the path name is fetched, and a callback is established for it if it is not already cached or if the client does not have a callback on it. Venus does lookups on the fetched directories locally, using fids. No requests are forwarded from one server to another. At the end of a path-name traversal, all the intermediate directories and the target file are in the cache with callbacks on them. Future open calls to this file will involve no network communication at all, unless a callback is broken on a component of the path name.

The only exception to the caching policy is a modification to a directory that is made directly on the server responsible for that directory for reasons of integrity. The Vice interface has well-defined operations for such purposes. Venus reflects the changes in its cached copy to avoid re-fetching the directory.

17.6.4 Implementation

Client processes are interfaced to a UNIX kernel with the usual set of system calls. The kernel is modified slightly to detect references to Vice files in the relevant operations and to forward the requests to the client-level Venus process at the workstation.

Venus carries out path-name translation component by component, as described above. It has a mapping cache that associates volumes to server locations in order to avoid server interrogation for an already known volume location. If a volume is not present in this cache, Venus contacts any server to which it already has a connection, requests the location information, and enters that information into the mapping cache. Unless Venus already has a connection to the server, it establishes a new connection. It then uses this connection to fetch the file or directory. Connection establishment is needed for authentication and security purposes. When a target file is found and cached, a copy is created on the local disk. Venus then returns to the kernel, which opens the cached copy and returns its handle to the client process.

The UNIX file system is used as a low-level storage system for both AFS servers and clients. The client cache is a local directory on the workstation's disk. Within this directory are files whose names are placeholders for cache entries. Both Venus and server processes access UNIX files directly by the latter's inodes to avoid the expensive path-name-to-inode translation routine (*namei*). Because the internal inode interface is not visible to client-level processes (both Venus and server processes are client-level processes), an appropriate set of additional system calls was added. DFS uses its own journaling file system to improve performance and reliability over UFS.

Venus manages two separate caches: one for status and the other for data. It uses a simple least-recently-used (LRU) algorithm to keep each of them bounded in size. When a file is flushed from the cache, Venus notifies the appropriate server to remove the callback for this file. The status cache is kept in virtual memory to allow rapid servicing of `stat()` (file-status-returning) system calls. The data cache is resident on the local disk, but the UNIX I/O buffering mechanism does some caching of disk blocks in memory that is transparent to Venus.

A single client-level process on each file server services all file requests from clients. This process uses a lightweight-process package with non-preemptible scheduling to service many client requests concurrently. The RPC package is integrated with the lightweight-process package, thereby allowing the file server to concurrently make or service one RPC per lightweight process. The RPC package is built on top of a low-level datagram abstraction. Whole-file transfer is implemented as a side effect of the RPC calls. One RPC connection exists per client, but there is no a priori binding of lightweight processes to these connections. Instead, a pool of lightweight processes services client requests on all connections. The use of a single multithreaded server process allows the caching of data structures needed to service requests. On the negative side, a crash of a single server process has the disastrous effect of paralyzing this particular server.

17.7 Summary

A DFS is a file-service system whose clients, servers, and storage devices are dispersed among the sites of a distributed system. Accordingly, service activity has to be carried out across the network; instead of a single centralized data repository, there are multiple independent storage devices.

Ideally, a DFS should look to its clients like a conventional, centralized file system. The multiplicity and dispersion of its servers and storage devices should be made transparent. That is, the client interface of a DFS should not distinguish between local and remote files. It is up to the DFS to locate the files and to arrange for the transport of the data. A transparent DFS facilitates client mobility by bringing the client's environment to the site where the client logs in.

There are several approaches to naming schemes in a DFS. In the simplest approach, files are named by some combination of their host name and local name, which guarantees a unique system-wide name. Another approach, popularized by NFS, provides a means to attach remote directories to local directories, thus giving the appearance of a coherent directory tree.

Requests to access a remote file are usually handled by two complementary methods. With remote service, requests for accesses are delivered to the server. The server machine performs the accesses, and their results are forwarded back to the client. With caching, if the data needed to satisfy the access request are not already cached, then a copy of the data is brought from the server to the client. Accesses are performed on the cached copy. The idea is to retain recently accessed disk blocks in the cache, so that repeated accesses to the same information can be handled locally, without additional network traffic. A replacement policy is used to keep the cache size bounded. The

problem of keeping the cached copies consistent with the master file is the cache-consistency problem.

There are two approaches to server-side information. Either the server tracks each file the client accesses, or it simply provides blocks as the client requests them without knowledge of their use. These approaches are the stateful versus stateless service paradigms.

Replication of files on different machines is a useful redundancy for improving availability. Multimachine replication can benefit performance, too, since selecting a nearby replica to serve an access request results in shorter service time.

AFS is a feature-rich DFS characterized by location independence and location transparency. It also imposes significant consistency semantics. Caching and replication are used to improve performance.

Exercises

17.1 What are the benefits of a DFS compared with a file system in a centralized system?

17.2 Which of the example DFSs discussed in this chapter would handle a large, multiclient database application most efficiently? Explain your answer.

17.3 Discuss whether AFS and NFS provide the following: (a) location transparency and (b) location independence.

17.4 Under what circumstances would a client prefer a location-transparent DFS? Under what circumstances would she prefer a location-independent DFS? Discuss the reasons for these preferences.

17.5 What aspects of a distributed system would you select for a system running on a totally reliable network?

17.6 Consider AFS, which is a stateful distributed file system. What actions need to be performed to recover from a server crash in order to preserve the consistency guaranteed by the system?

17.7 Compare and contrast the techniques of caching disk blocks locally, on a client system, and remotely, on a server.

17.8 AFS is designed to support a large number of clients. Discuss three techniques used to make AFS a scalable system.

17.9 Discuss the advantages and disadvantages of performing path-name translation by having the client ship the entire path to the server requesting a translation for the entire path name of the file.

17.10 What are the benefits of mapping objects into virtual memory, as Apollo Domain does? What are the drawbacks?

17.11 Describe some of the fundamental differences between AFS and NFS (see Chapter 11).

17.12 Discuss whether clients in the following systems can obtain inconsistent or stale data from the file server and, if so, under what scenarios this could occur.

a. AFS

b. Sprite

c. NFS

Bibliographical Notes

Discussions concerning consistency and recovery control for replicated files were offered by Davcev and Burkhard [1985]. Management of replicated files in a UNIX environment was covered by Brereton [1986] and Purdin et al. [1987]. Wah [1984] discussed the issue of file placement on distributed computer systems. A detailed survey of mainly centralized file servers was given in Svobodova [1984].

Sun's network file system (NFS) was presented by Callaghan [2000] and Sandberg et al. [1985]. The AFS system was discussed by Morris et al. [1986], Howard et al. [1988], and Satyanarayanan [1990]. Information about OpenAFS is available from http://www.openafs.org

Many different and interesting DFSs were not covered in detail in this text, including UNIX United, Sprite, and Locus. UNIX United was described by Brownbridge et al. [1982]. The Locus system was discussed by Popek and Walker [1985]. The Sprite system was described by Ousterhout et al. [1988] and Nelson et al. [1988]. Distributed file systems for mobile storage devices were discussed in Kistler and Satyanarayanan [1992] and Sobti et al. [2004]. Considerable research has also been performed on cluster-based distributed file systems (Anderson et al. [1995], Lee and Thekkath [1996], Thekkath et al. [1997], and Anderson et al. [2000]). Distributed storage systems for large-scale, wide-area settings were presented in Dabek et al. [2001] and Kubiatowicz et al. [2000].

Distributed Coordination



In Chapter 6, we described various mechanisms that allow processes to synchronize their actions. We also discussed a number of schemes to ensure the atomicity of a transaction that executes either in isolation or concurrently with other transactions. In Chapter 7, we described various methods that an operating system can use to deal with the deadlock problem. In this chapter, we examine how centralized synchronization mechanisms can be extended to a distributed environment. We also discuss methods for handling deadlocks in a distributed system.

CHAPTER OBJECTIVES

- To describe various methods for achieving mutual exclusion in a distributed system.
- To explain how atomic transactions can be implemented in a distributed system.
- To show how some of the concurrency-control schemes discussed in Chapter 6 can be modified for use in a distributed environment.
- To present schemes for handling deadlock prevention, deadlock avoidance, and deadlock detection in a distributed system.

18.1 Event Ordering

In a centralized system, we can always determine the order in which two events occurred, since the system has a single common memory and clock. Many applications may require us to determine order. For example, in a resource-allocation scheme, we specify that a resource can be used only *after* the resource has been granted. A distributed system, however, has no common memory and no common clock. Therefore, it is sometimes impossible to say which of two events occurred first. The *happened-before* relation is only a partial ordering of the events in distributed systems. Since the ability to define a total ordering is

crucial in many applications, we present a distributed algorithm for extending the *happened-before* relation to a consistent total ordering of all the events in the system.

18.1.1 The Happened-Before Relation

Since we are considering only sequential processes, all events executed in a single process are totally ordered. Also, by the law of causality, a message can be received only after it has been sent. Therefore, we can define the *happened-before* relation (denoted by $\rightarrow$) on a set of events as follows (assuming that sending and receiving a message constitutes an event):

1. If A and B are events in the same process, and A was executed before B, then A $\rightarrow$ B.
2. If A is the event of sending a message by one process and B is the event of receiving that message by another process, then A $\rightarrow$ B.
3. If A $\rightarrow$ B and B $\rightarrow$ C then A $\rightarrow$ C.

Since an event cannot happen before itself, the $\rightarrow$ relation is an irreflexive partial ordering.

If two events, A and B, are not related by the $\rightarrow$ relation (that is, A did not happen before B, and B did not happen before A), then we say that these two events were executed **concurrently**. In this case, neither event can causally affect the other. If, however, A $\rightarrow$ B, then it is possible for event A to affect event B causally.

A space–time diagram, such as that in Figure 18.1, can best illustrate the definitions of concurrency and *happened-before*. The horizontal direction represents space (that is, different processes), and the vertical direction represents time. The labeled vertical lines denote processes (or processors). The labeled dots denote events. A wavy line denotes a message sent from one process to another. Events are concurrent if and only if no path exists between them.

For example, these are some of the events related by the *happened-before* relation in Figure 18.1:

$$p_1 \rightarrow q_2$$
$$r_0 \rightarrow q_4$$
$$q_3 \rightarrow r_4$$
$$p_1 \rightarrow q_4 \text{ (since } p_1 \rightarrow q_2 \text{ and } q_2 \rightarrow q_4\text{)}$$

These are some of the concurrent events in the system:

$$q_0 \text{ and } p_2$$
$$r_0 \text{ and } q_3$$
$$r_0 \text{ and } p_3$$
$$q_3 \text{ and } p_3$$

We cannot know which of two concurrent events, such as q_0 and p_2, happened first. However, since neither event can affect the other (there is no way for one of them to know whether the other has occurred yet), it is not important which

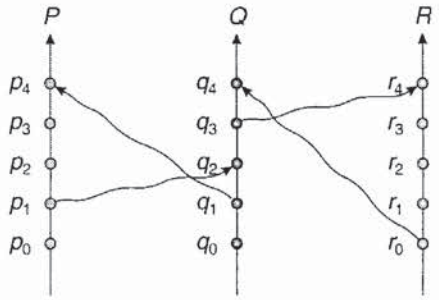


Figure 18.1 Relative time for three concurrent processes.

happened first. It is important only that any processes that care about the order of two concurrent events agree on some order.

18.1.2 Implementation

To determine that an event A happened before an event B, we need either a common clock or a set of perfectly synchronized clocks. Since neither of these is available in a distributed system, we must define the *happened-before* relation without the use of physical clocks.

We associate with each system event a **timestamp**. We can then define the **global ordering** requirement: For every pair of events A and B, if A $\rightarrow$ B, then the timestamp of A is less than the timestamp of B. (Below, we will see that the converse need not be true.)

How do we enforce the global ordering requirement in a distributed environment? We define within *each* process P_i a **logical clock**, LC_i. The logical clock can be implemented as a simple counter incremented between any two successive events executed within a process. Since the logical clock has a **monotonically** increasing value, it assigns a unique number to every event, and if an event A occurs before event B in process P_i, then $LC_i(A) < LC_i(B)$. The timestamp for an event is the value of the logical clock for that event. This scheme ensures that for any two events in the same process the global ordering requirement is met.

Unfortunately, this scheme does not ensure that the global ordering requirement is met across processes. To illustrate the problem, consider two processes P_1 and P_2 that communicate with each other. Suppose that P_1 sends a message to P_2 (event A) with $LC_1(A) = 200$, and P_2 receives the message (event B) with $LC_2(B) = 195$ (because the processor for P_2 is slower than the processor for P_1, its logical clock ticks more slowly). This situation violates our requirement, since A $\rightarrow$ B but the timestamp of A is greater than the timestamp of B.

To resolve this difficulty, we require a process to advance its logical clock when it receives a message whose timestamp is greater than the current value of its logical clock. In particular, if process P_i receives a message (event B) with timestamp t and $LC_i(B) \leq t$, then it should advance its clock so that $LC_i(B) = t + 1$. Thus, in our example, when P_2 receives the message from P_1, it will advance its logical clock so that $LC_2(B) = 201$.

Finally, to realize a total ordering, we need only observe that, with our timestamp-ordering scheme, if the timestamps of two events, A and B, are the same, then the events are concurrent. In this case, we may use process identity numbers to break ties and to create a total ordering. The use of timestamps is further discussed in Section 18.4.2.

18.2 Mutual Exclusion

In this section, we present a number of different algorithms for implementing mutual exclusion in a distributed environment. We assume that the system consists of n processes, each of which resides at a different processor. To simplify our discussion, we assume that processes are numbered uniquely from 1 to n and that a one-to-one mapping exists between processes and processors (that is, each process has its own processor).

18.2.1 Centralized Approach

In a centralized approach to providing mutual exclusion, one of the processes in the system is chosen to coordinate the entry to the critical section. Each process that wants to invoke mutual exclusion sends a *request* message to the coordinator. When the process receives a *reply* message from the coordinator, it can proceed to enter its critical section. After exiting its critical section, the process sends a *release* message to the coordinator and proceeds with its execution.

On receiving a *request* message, the coordinator checks to see whether some other process is in its critical section. If no process is in its critical section, the coordinator immediately sends back a *reply* message. Otherwise, the request is queued. When the coordinator receives a *release* message, it removes one of the *request* messages from the queue (in accordance with some scheduling algorithm) and sends a *reply* message to the requesting process.

It should be clear that this algorithm ensures mutual exclusion. In addition, if the scheduling policy within the coordinator is fair—such as first-come, first-served (FCFS) scheduling—no starvation can occur. This scheme requires three messages per critical-section entry: a *request*, a *reply*, and a *release*.

If the coordinator process fails, then a new process must take its place. In Section 18.6, we describe some algorithms for electing a unique new coordinator. Once a new coordinator has been elected, it must poll all the processes in the system to reconstruct its *request* queue. Once the queue has been constructed, the computation can resume.

18.2.2 Fully Distributed Approach

If we want to distribute the decision making across the entire system, then the solution is far more complicated. One approach, described next, uses an algorithm based on the event-ordering scheme described in Section 18.1.

When a process P_i wants to enter its critical section, it generates a new timestamp, TS, and sends the message *request*(P_i, TS) to all processes in the system (including itself). On receiving a *request* message, a process may reply immediately (that is, send a *reply* message back to P_i), or it may defer sending a reply back (because it is already in its critical section, for example). A process

that has received a *reply* message from all other processes in the system can enter its critical section, queueing incoming requests and deferring them. After exiting its critical section, the process sends *reply* messages to all its deferred requests.

The decision whether process P_i replies immediately to a *request*(P_j, TS) message or defers its reply is based on three factors:

1. If process P_i is in its critical section, then it defers its reply to P_j.
2. If process P_i does *not* want to enter its critical section, then it sends a reply immediately to P_j.
3. If process P_i wants to enter its critical section but has not yet entered it, then it compares its own *request* timestamp with the timestamp of the incoming request made by process P_j. If its own *request* timestamp is greater than that of the incoming request, then it sends a reply immediately to P_j (P_j asked first). Otherwise, the reply is deferred.

This algorithm exhibits the following desirable behavior:

- Mutual exclusion is obtained.
- Freedom from deadlock is ensured.
- Freedom from starvation is ensured, since entry to the critical section is scheduled according to the timestamp ordering. The timestamp ordering ensures that processes are served in FCFS order.
- The number of messages per critical-section entry is $2 \times (n - 1)$. This number is the minimum number of required messages per critical-section entry when processes act independently and concurrently.

To illustrate how the algorithm functions, we consider a system consisting of processes P_1, P_2, and P_3. Suppose that processes P_1 and P_3 want to enter their critical sections. Process P_1 then sends a message *request* (P_1, timestamp = 10) to processes P_2 and P_3, while process P_3 sends a message *request* (P_3, timestamp = 4) to processes P_1 and P_2. The timestamps 4 and 10 were obtained from the logical clocks described in Section 18.1. When process P_2 receives these *request* messages, it replies immediately. When process P_1 receives the *request* from process P_3, it replies immediately, since the timestamp (10) on its own *request* message is greater than the timestamp (4) for process P_3. When process P_3 receives the *request* message from process P_1, it defers its reply, since the timestamp (4) on its *request* message is less than the timestamp (10) for the message from process P_1. On receiving replies from both process P_1 and process P_2, process P_3 can enter its critical section. After exiting its critical section, process P_3 sends a reply to process P_1, which can then enter its critical section.

Because this scheme requires the participation of all the processes in the system, however, it has three undesirable consequences:

1. The processes need to know the identity of all other processes in the system. When a new process joins the group of processes participating in the mutual-exclusion algorithm, the following actions need to be taken:

a. The process must receive the names of all the other processes in the group.

b. The name of the new process must be distributed to all the other processes in the group.

This task is not as trivial as it may seem, since some *request* and *reply* messages may be circulating in the system when the new process joins the group. The interested reader is referred to the Bibliographical Notes for more details.

2. If one process fails, then the entire scheme collapses. We can resolve this difficulty by continuously monitoring the state of all processes in the system. If one process fails, then all other processes are notified, so that they will no longer send *request* messages to the failed process. When a process recovers, it must initiate the procedure that allows it to rejoin the group.

3. Processes that have not entered their critical section must pause frequently to assure other processes that they intend to enter the critical section.

Because of these difficulties, this protocol is best suited for small, stable sets of cooperating processes.

18.2.3 Token-Passing Approach

Another method of providing mutual exclusion is to circulate a token among the processes in the system. A **token** is a special type of message that is passed around the system. Possession of the token entitles the holder to enter the critical section. Since there is only a single token, only one process can be in its critical section at a time.

We assume that the processes in the system are *logically* organized in a **ring structure.** The physical communication network need not be a ring. As long as the processes are connected to one another, it is possible to implement a logical ring. To implement mutual exclusion, we pass the token around the ring. When a process receives the token, it may enter its critical section, keeping the token. After the process exits its critical section, the token is passed around again. If the process receiving the token does not want to enter its critical section, it passes the token to its neighbor. This scheme is similar to algorithm 1 in Chapter 6, but a token is substituted for a shared variable.

If the ring is unidirectional, freedom from starvation is ensured. The number of messages required to implement mutual exclusion may vary from one message per entry, in the case of high contention (that is, every process wants to enter its critical section), to an infinite number of messages, in the case of low contention (that is, no process wants to enter its critical section).

Two types of failure must be considered. First, if the token is lost, an election must be called to generate a new token. Second, if a process fails, a new logical ring must be established. In Section 18.6, we present an election algorithm; others are possible. The development of an algorithm for reconstructing the ring is left to you in Exercise 18.9.

18.3 Atomicity

In Chapter 6, we introduced the concept of an atomic transaction, which is a program unit that must be executed **atomically**. That is, either all the operations associated with it are executed to completion, or none are performed. When we are dealing with a distributed system, ensuring the atomicity of a transaction becomes much more complicated than in a centralized system. This difficulty occurs because several sites may be participating in the execution of a single transaction. The failure of one of these sites, or the failure of a communication link connecting the sites, may result in erroneous computations.

Ensuring that the execution of transactions in the distributed system preserves atomicity is the function of the **transaction coordinator**. Each site has its own local transaction coordinator, which is responsible for coordinating the execution of all the transactions initiated at that site. For each such transaction, the coordinator is responsible for the following:

- Starting the execution of the transaction
- Breaking the transaction into a number of subtransactions and distributing these subtransactions to the appropriate sites for execution
- Coordinating the termination of the transaction, which may result in the transactions being committed at all sites or aborted at all sites

We assume that each local site maintains a log for recovery purposes.

18.3.1 The Two-Phase Commit Protocol

For atomicity to be ensured, all the sites in which a transaction T has executed must agree on the final outcome of the execution. T must either commit at all sites, or it must abort at all sites. To ensure this property, the transaction coordinator of T must execute a **commit protocol**. Among the simplest and most widely used commit protocols is the **two-phase commit (2PC) protocol**, which we discuss next.

Let T be a transaction initiated at site S_i, and let the transaction coordinator at S_i be C_i. When T completes its execution—that is, when all the sites at which T has executed inform C_i that T has completed—then C_i starts the 2PC protocol.

- **Phase 1.** C_i adds the record <prepare T> to the log and forces the record onto stable storage. It then sends a *prepare (T)* message to all the sites at which T has executed. On receiving the message, the transaction manager at that site determines whether it is willing to commit its portion of T. If the answer is *no*, it adds a record <no T> to the log, and then it responds by sending an *abort (T)* message to C_i. If the answer is *yes*, it adds a record <ready T> to the log and forces all the log records corresponding to T onto stable storage. The transaction manager then replies with a *ready (T)* message to C_i.
- **Phase 2.** When C_i has received responses to the *prepare (T)* message from all the sites, or when a pre-specified interval of time has elapsed since the *prepare (T)* message was sent out, C_i can determine whether the transaction

T can be committed or aborted. Transaction T can be committed if C_i has received a *ready (T)* message from all the participating sites. Otherwise, transaction T must be aborted. Depending on the verdict, either a record <commit T> or a record <abort T> is added to the log and is forced onto stable storage. At this point, the fate of the transaction has been sealed. Following this, the coordinator sends either a *commit (T)* or an *abort (T)* message to all participating sites. When a site receives that message, it records the message in the log.

A site at which T has executed can unconditionally abort T at any time prior to its sending the message *ready (T)* to the coordinator. The *ready (T)* message is, in effect, a promise by a site to follow the coordinator's order to commit T or to abort T. A site can make such a promise only when the needed information is stored in stable storage. Otherwise, if the site crashes after sending ready T, it may be unable to make good on its promise.

Since unanimity is required to commit a transaction, the fate of T is sealed as soon as at least one site responds with *abort (T)*. Note that the coordinator site S_i can decide unilaterally to abort T, as it is one of the sites at which T has executed. The final verdict regarding T is determined at the time the coordinator writes that verdict (commit or abort) to the log and forces it to stable storage. In some implementations of the 2PC protocol, a site sends an *acknowledge (T)* message to the coordinator at the end of the second phase of the protocol. When the coordinator has received the *acknowledge (T)* message from all the sites, it adds the record <complete T> to the log.

18.3.2 Failure Handling in 2PC

We now examine in detail how 2PC responds to various types of failures. As we shall see, one major disadvantage of the 2PC protocol is that coordinator failure may result in blocking, and a decision either to commit or to abort T may have to be postponed until C_i recovers.

18.3.2.1 Failure of a Participating Site

When a participating site S_k recovers from a failure, it must examine its log to determine the fate of those transactions that were in the midst of execution when the failure occurred. Let T be one such transaction. How will S_k deal with T? We consider each of the possible alternatives:

- The log contains a <commit T> record. In this case, the site executes `redo`(T).
- The log contains an <abort T> record. In this case, the site executes `undo`(T).
- The log contains a <ready T> record. In this case, the site must consult C_i to determine the fate of T. If C_i is up, it notifies S_k regarding whether T committed or aborted. In the former case, it executes `redo`(T); in the latter case, it executes `undo`(T). If C_i is down, S_k must try to find out the fate of T from other sites. It does so by sending a *query-status (T)* message to all

the sites in the system. On receiving such a message, a site must consult its log to determine whether T has executed there and, if so, whether T committed or aborted. It then notifies S_k about this outcome. If no site has the appropriate information (that is, whether T committed or aborted), then S_k can neither abort nor commit T. The decision concerning T is postponed until S_k can obtain the needed information. Thus, S_k must periodically resend the *query-status (T)* message to the other sites. It does so until a site recovers that contains the needed information. The site at which C_i resides always has the needed information.

- The log contains no control records (abort, commit, ready) concerning T. The absence of control records implies that S_k failed before responding to the *prepare (T)* message from C_i. Since the failure of S_k means that it could not have sent such a response, by our algorithm, C_i must have aborted T. Hence, S_k must execute undo(T).

18.3.2.2 Failure of the Coordinator

If the coordinator fails in the midst of the execution of the commit protocol for transaction T, then the participating sites must decide on the fate of T. We shall see that, in certain cases, the participating sites cannot decide whether to commit or abort T, and therefore these sites must wait for the recovery of the failed coordinator.

- If an active site contains a <commit T> record in its log, then T must be committed.
- If an active site contains an <abort T> record in its log, then T must be aborted.
- If some active site does *not* contain a <ready T> record in its log, then the failed coordinator C_i cannot have decided to commit T. We can draw this conclusion because a site that does not have a <ready T> record in its log cannot have sent a *ready (T)* message to C_i. However, the coordinator may have decided to abort T. Rather than wait for C_i to recover, it is preferable to abort T in this case.
- If none of the preceding cases holds, then all the active sites must have a <ready T> record in their logs, but no additional control records (such as <abort T> or <commit T>). Since the coordinator has failed, it is impossible to determine whether a decision has been made—or, if so, what that decision is—until the coordinator recovers. Thus, the active sites must wait for C_i to recover. As long as the fate of T remains in doubt, T may continue to hold system resources. For example, if locking is used, T may hold locks on data at active sites. Such a situation is undesirable because hours or days may pass before C_i is again active. During this time, other transactions may be forced to wait for T. As a result, data are unavailable not only on the failed site (C_i) but on active sites as well. The amount of unavailable data increases as the downtime of C_i grows. This situation is called the *blocking* problem, because T is blocked pending the recovery of site C_i.

18.3.2.3 Failure of the Network

When a link fails, the messages in the process of being routed through the link do not arrive at their destinations intact. From the viewpoint of the sites connected throughout that link, the other sites appear to have failed. Thus, our previous schemes apply here as well.

When a number of links fail, the network may partition. In this case, two possibilities exist. The coordinator and all its participants may remain in one partition; in this case, the failure has no effect on the commit protocol. Alternatively, the coordinator and its participants may belong to several partitions; in this case, messages between the participant and the coordinator are lost, reducing the case to a link failure.

18.4 Concurrency Control

We move next to the issue of concurrency control. In this section, we show how certain of the concurrency-control schemes discussed in Chapter 6 can be modified for use in a distributed environment.

The transaction manager of a distributed database system manages the execution of those transactions (or subtransactions) that access data stored in a local site. Each such transaction may be either a local transaction (that is, a transaction that executes only at that site) or part of a global transaction (that is, a transaction that executes at several sites). Each transaction manager is responsible for maintaining a log for recovery purposes and for participating in an appropriate concurrency-control scheme to coordinate the concurrent execution of the transactions executing at that site. As we shall see, the concurrency schemes described in Chapter 6 need to be modified to accommodate the distribution of transactions.

18.4.1 Locking Protocols

The two-phase locking protocols described in Chapter 6 can be used in a distributed environment. The only change needed is in the way the lock manager is implemented. Here, we present several possible schemes. The first deals with the case where no data replication is allowed. The others apply to the more general case where data can be replicated in several sites. As in Chapter 6, we assume the existence of the **shared** and **exclusive lock modes**.

18.4.1.1 Nonreplicated Scheme

If no data are replicated in the system, then the locking schemes described in Section 6.9 can be applied as follows: Each site maintains a local lock manager whose function is to administer the lock and unlock requests for those data items stored in that site. When a transaction wishes to lock data item Q at site S_i, it simply sends a message to the lock manager at site S_i requesting a lock (in a particular lock mode). If data item Q is locked in an incompatible mode, then the request is delayed until that request can be granted. Once it has been determined that the lock request can be granted, the lock manager sends a message back to the initiator indicating that the lock request has been granted.

This scheme has the advantage of simple implementation. It requires two message transfers for handling lock requests and one message transfer for handling unlock requests. However, deadlock handling is more complex. Since the lock and unlock requests are no longer made at a single site, the various deadlock-handling algorithms discussed in Chapter 7 must be modified; these modifications are discussed in Section 18.5.

18.4.1.2 Single-Coordinator Approach

Several concurrency-control schemes can be used in systems that allow data replication. Under the single-coordinator approach, the system maintains a *single* lock manager that resides in a *single* chosen site—say, S_i. All lock and unlock requests are made at site S_i. When a transaction needs to lock a data item, it sends a lock request to S_i. The lock manager determines whether the lock can be granted immediately. If so, it sends a message to that effect to the site at which the lock request was initiated. Otherwise, the request is delayed until it can be granted; and at that time, a message is sent to the site at which the lock request was initiated. The transaction can read the data item from *any* one of the sites at which a replica of the data item resides. In the case of a `write` operation, all the sites where a replica of the data item resides must be involved in the writing.

The scheme has the following advantages:

- **Simple implementation**. This scheme requires two messages for handling lock requests and one message for handling unlock requests.
- **Simple deadlock handling**. Since all lock and unlock requests are made at one site, the deadlock-handling algorithms discussed in Chapter 7 can be applied directly to this environment.

The disadvantages of the scheme include the following:

- **Bottleneck**. The site S_i becomes a bottleneck, since all requests must be processed there.
- **Vulnerability**. If the site S_i fails, the concurrency controller is lost. Either processing must stop or a recovery scheme must be used.

A compromise between these advantages and disadvantages can be achieved through a **multiple-coordinator approach**, in which the lock-manager function is distributed over several sites. Each lock manager administers the lock and unlock requests for a subset of the data items, and the lock managers reside in different sites. This distribution reduces the degree to which the coordinator is a bottleneck, but it complicates deadlock handling, since the lock and unlock requests are not made at a single site.

18.4.1.3 Majority Protocol

The majority protocol is a modification of the nonreplicated data scheme presented earlier. The system maintains a lock manager at each site. Each manager controls the locks for all the data or replicas of data stored at that site. When a transaction wishes to lock a data item Q that is replicated in n different

sites, it must send a lock request to more than one-half of the n sites in which Q is stored. Each lock manager determines whether the lock can be granted immediately (as far as it is concerned). As before, the response is delayed until the request can be granted. The transaction does not operate on Q until it has successfully obtained a lock on a majority of the replicas of *ch18/18*.

This scheme deals with replicated data in a decentralized manner, thus avoiding the drawbacks of central control. However, it suffers from its own disadvantages:

- **Implementation**. The majority protocol is more complicated to implement than the previous schemes. It requires $2(n/2 + 1)$ messages for handling lock requests and $(n/2 + 1)$ messages for handling unlock requests.
- **Deadlock handling**. Since the lock and unlock requests are not made at one site, the deadlock-handling algorithms must be modified (Section 18.5). In addition, a deadlock can occur even if only one data item is being locked. To illustrate, consider a system with four sites and full replication. Suppose that transactions T_1 and T_2 wish to lock data item Q in exclusive mode. Transaction T_1 may succeed in locking Q at sites S_1 and S_3, while transaction T_2 may succeed in locking Q at sites S_2 and S_4. Each then must wait to acquire the third lock, and hence a deadlock has occurred.

18.4.1.4 Biased Protocol

The biased protocol is similar to the majority protocol. The difference is that requests for shared locks are given more favorable treatment than are requests for exclusive locks. The system maintains a lock manager at each site. Each manager manages the locks for all the data items stored at that site. Shared and exclusive locks are handled differently.

- **Shared locks**. When a transaction needs to lock data item Q, it simply requests a lock on Q from the lock manager at one site containing a replica of *ch18/18*.
- **Exclusive locks**. When a transaction needs to lock data item Q, it requests a lock on Q from the lock manager at each site containing a replica of *ch18/18*.

As before, the response to the request is delayed until the request can be granted.

The scheme has the advantage of imposing less overhead on read operations than does the majority protocol. This advantage is especially significant in common cases in which the frequency of reads is much greater than the frequency of writes. However, the additional overhead on writes is a disadvantage. Furthermore, the biased protocol shares the majority protocol's disadvantage of complexity in handling deadlock.

18.4.1.5 Primary Copy

Yet another alternative is to choose one of the replicas as the primary copy. Thus, for each data item Q, the primary copy of Q must reside in precisely one site, which we call the *primary site of* Q. When a transaction needs to lock a data

item Q, it requests a lock at the primary site of *ch18/18*. As before, the response to the request is delayed until the request can be granted.

This scheme enables us to handle concurrency control for replicated data in much the same way as for unreplicated data. Implementation of the method is simple. However, if the primary site of Q fails, Q is inaccessible even though other sites containing a replica may be accessible.

18.4.2 Timestamping

The principal idea behind the timestamping scheme discussed in Section 6.9 is that each transaction is given a *unique* timestamp, which is used to decide the serialization order. Our first task, then, in generalizing the centralized scheme to a distributed scheme is to develop a method for generating unique timestamps. Our previous protocols can then be applied directly to the nonreplicated environment.

18.4.2.1 Generation of Unique Timestamps

Two primary methods are used to generate unique timestamps; one is centralized, and one is distributed. In the centralized scheme, a single site is chosen for distributing the timestamps. The site can use a logical counter or its own local clock for this purpose.

In the distributed scheme, each site generates a local unique timestamp using either a logical counter or the local clock. The global unique timestamp is obtained by concatenation of the local unique timestamp with the site identifier, which must be unique (Figure 18.2). The order of concatenation is important! We use the site identifier in the least significant position to ensure that the global timestamps generated in one site are not always greater than those generated in another site. Compare this technique for generating unique timestamps with the one we presented in Section 18.1.2 for generating unique names.

We may still have a problem if one site generates local timestamps at a faster rate than do other sites. In such a case, the fast site's logical counter will be larger than those of other sites. Therefore, all timestamps generated by the fast site will be larger than those generated by other sites. A mechanism is needed to ensure that local timestamps are generated fairly across the system. To accomplish the fair generation of timestamps, we define within each site S_i a logical clock (LC_i), which generates the local timestamp (see Section 18.1.2). To ensure that the various logical clocks are synchronized, we require that a site S_i advance its logical clock whenever a transaction T_i with timestamp $<x,y>$

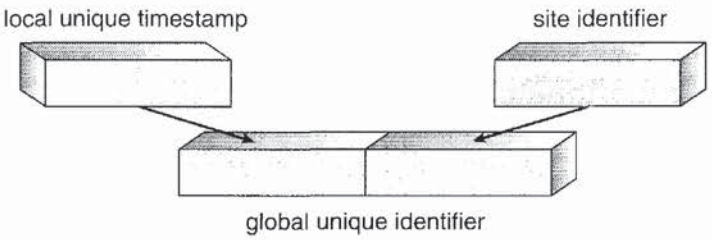


Figure 18.2 Generation of unique timestamps.

visits that site and x is greater than the current value of LC_i. In this case, site S_i advances its logical clock to the value $x + 1$.

If the system clock is used to generate timestamps, then timestamps are assigned fairly, provided that no site has a system clock that runs fast or slow. Since clocks may not be perfectly accurate, a technique similar to that used for logical clocks must be used to ensure that no clock gets far ahead or far behind another clock.

18.4.2.2 Timestamp-Ordering Scheme

The basic timestamp scheme introduced in Section 6.9 can be extended in a straightforward manner to a distributed system. As in the centralized case, cascading rollbacks may result if no mechanism is used to prevent a transaction from reading a data item value that is not yet committed. To eliminate cascading rollbacks, we can combine the basic timestamp scheme of Section 6.9 with the 2PC protocol of Section 18.3 to obtain a protocol that ensures serializability with no cascading rollbacks. We leave the development of such an algorithm to you.

The basic timestamp scheme just described suffers from the undesirable property that conflicts between transactions are resolved through rollbacks, rather than through waits. To alleviate this problem, we can buffer the various `read` and `write` operations (that is, *delay* them) until a time when we are assured that these operations can take place without causing aborts. A `read`(x) operation by T_i must be delayed if there exists a transaction T_j that will perform a `write`(x) operation but has not yet done so and $\text{TS}(T_j) < \text{TS}(T_i)$. Similarly, a `write`($x$) operation by T_i must be delayed if there exists a transaction T_j that will perform either a `read`(x) or a `write`(x) operation and $\text{TS}(T_j) < \text{TS}(T_i)$. Various methods are available for ensuring this property. One such method, called the **conservative timestamp-ordering scheme**, requires each site to maintain a read queue and a write queue consisting of all the `read` and `write` requests that are to be executed at the site and that must be delayed to preserve the above property. We shall not present the scheme here. Again, we leave the development of the algorithm to you.

18.5 Deadlock Handling

The deadlock-prevention, deadlock-avoidance, and deadlock-detection algorithms presented in Chapter 7 can be extended so that they can be used in a distributed system. In this section, we describe several of these distributed algorithms.

18.5.1 Deadlock Prevention and Avoidance

The deadlock-prevention and deadlock-avoidance algorithms presented in Chapter 7 can be used in a distributed system, provided that appropriate modifications are made. For example, we can use the resource-ordering deadlock-prevention technique by simply defining a global ordering among the system resources. That is, all resources in the entire system are assigned unique numbers, and a process may request a resource (at any processor) with

unique number i only if it is not holding a resource with a unique number greater than i. Similarly, we can use the banker's algorithm in a distributed system by designating one of the processes in the system (the *banker*) as the process that maintains the information necessary to carry out the banker's algorithm. Every resource request must be channelled through the banker.

The global resource-ordering deadlock-prevention scheme is simple to implement in a distributed environment and requires little overhead. The banker's algorithm can also be implemented easily, but it may require too much overhead. The banker may become a bottleneck, since the number of messages to and from the banker may be large. Thus, the banker's scheme does not seem to be of practical use in a distributed system.

We turn next to a new deadlock-prevention scheme based on a timestamp-ordering approach with resource preemption. Although this approach can handle any deadlock situation that may arise in a distributed system, for simplicity we consider only the case of a single instance of each resource type.

To control the preemption, we assign a unique priority number to each process. These numbers are used to decide whether a process P_i should wait for a process P_j. For example, we can let P_i wait for P_j if P_i has a priority higher than that of P_j; otherwise, P_i is rolled back. This scheme prevents deadlocks because, for every edge $P_i \rightarrow P_j$ in the wait-for graph, P_i has a higher priority than P_j. Thus, a cycle cannot exist.

One difficulty with this scheme is the possibility of starvation. Some processes with extremely low priorities may always be rolled back. This difficulty can be avoided through the use of timestamps. Each process in the system is assigned a unique timestamp when it is created. Two complementary deadlock-prevention schemes using timestamps have been proposed:

1. **The wait–die scheme**. This approach is based on a nonpreemptive technique. When process P_i requests a resource currently held by P_j, P_i is allowed to wait only if it has a smaller timestamp than does P_j (that is, P_i is older than P_j). Otherwise, P_i is rolled back (dies). For example, suppose that processes P_1, P_2, and P_3 have timestamps 5, 10, and 15, respectively. If P_1 requests a resource held by P_2, P_1 will wait. If P_3 requests a resource held by P_2, P_3 will be rolled back.

2. **The wound–wait scheme**. This approach is based on a preemptive technique and is a counterpart to the wait–die approach. When process P_i requests a resource currently held by P_j, P_i is allowed to wait only if it has a larger timestamp than does P_j (that is, P_i is younger than P_j). Otherwise, P_j is rolled back (P_j is *wounded* by P_i). Returning to our previous example, with processes P_1, P_2, and P_3, if P_1 requests a resource held by P_2, then the resource will be preempted from P_2, and P_2 will be rolled back. If P_3 requests a resource held by P_2, then P_3 will wait.

Both schemes can avoid starvation provided that, when a process is rolled back, it is *not* assigned a new timestamp. Since timestamps always increase, a process that is rolled back will eventually have the smallest timestamp. Thus, it will not be rolled back again. There are, however, significant differences in the way the two schemes operate.

- In the wait–die scheme, an older process must wait for a younger one to release its resource. Thus, the older the process gets, the more it tends to wait. By contrast, in the wound–wait scheme, an older process never waits for a younger process.
- In the wait–die scheme, if a process P_i dies and is rolled back because it has requested a resource held by process P_j, then P_i may reissue the same sequence of requests when it is restarted. If the resource is still held by P_j, then P_i will die again. Thus, P_i may die several times before acquiring the needed resource. Contrast this series of events with what happens in the wound–wait scheme. Process P_i is wounded and rolled back because P_j has requested a resource it holds. When P_i is restarted and requests the resource now being held by P_j, P_i waits. Thus, fewer rollbacks occur in the wound–wait scheme.

The major problem with both schemes is that unnecessary rollbacks may occur.

18.5.2 Deadlock Detection

The deadlock-prevention algorithm may preempt resources even if no deadlock has occurred. To prevent unnecessary preemptions, we can use a deadlock-detection algorithm. We construct a wait-for graph describing the resource-allocation state. Since we are assuming only a single resource of each type, a cycle in the wait-for graph represents a deadlock.

The main problem in a distributed system is deciding how to maintain the wait-for graph. We illustrate this problem by describing several common techniques to deal with this issue. These schemes require each site to keep a *local* wait-for graph. The nodes of the graph correspond to all the processes (local as well as nonlocal) currently holding or requesting any of the resources local to that site. For example, in Figure 18.3 we have a system consisting of two sites, each maintaining its local wait-for graph. Note that processes P_2 and P_3 appear in both graphs, indicating that the processes have requested resources at both sites.

These local wait-for graphs are constructed in the usual manner for local processes and resources. When a process P_i in site S_1 needs a resource held by process P_j in site S_2, a request message is sent by P_i to site S_2. The edge $P_i \rightarrow P_j$ is then inserted in the local wait-for graph of site S_2.

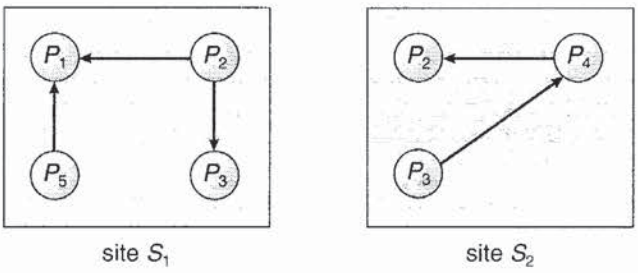


Figure 18.3 Two local wait-for graphs.

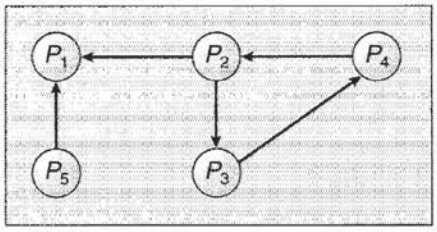


Figure 18.4 Global wait-for graph for Figure 18.3.

Clearly, if any local wait-for graph has a cycle, deadlock has occurred. The fact that we find no cycles in any of the local wait-for graphs does not mean that there are no deadlocks, however. To illustrate this problem, we consider the system depicted in Figure 18.3. Each wait-for graph is acyclic; nevertheless, a deadlock exists in the system. To prove that a deadlock has not occurred, we must show that the **union** of all local graphs is acyclic. The graph (Figure 18.4) that we obtain by taking the union of the two wait-for graphs of Figure 18.3 does indeed contain a cycle, implying that the system is in a deadlocked state.

A number of methods are available to organize the wait-for graph in a distributed system. We describe several common schemes here.

18.5.2.1 Centralized Approach

In the centralized approach, a global wait-for graph is constructed as the union of all the local wait-for graphs. It is maintained in a *single* process: the **deadlock-detection coordinator**. Since there is communication delay in the system, we must distinguish between two types of wait-for graphs. The *real* graph describes the real but unknown state of the system at any instance in time, as would be seen by an omniscient observer. The *constructed* graph is an approximation generated by the coordinator during the execution of its algorithm. The constructed graph must be generated so that, whenever the detection algorithm is invoked, the reported results are correct. By *correct* we mean the following:

- If a deadlock exists, then it is reported properly.
- If a deadlock is reported, then the system is indeed in a deadlocked state.

As we shall show, it is not easy to construct such correct algorithms.

The wait-for graph may be constructed at three different points in time:

1. Whenever a new edge is inserted in or removed from one of the local wait-for graphs
2. Periodically, when a number of changes have occurred in a wait-for graph
3. Whenever the deadlock-detection coordinator needs to invoke the cycle-detection algorithm

When the deadlock-detection algorithm is invoked, the coordinator searches its global graph. If a cycle is found, a *victim* is selected to be rolled back. The

coordinator must notify all the sites that a particular process has been selected as victim. The sites, in turn, roll back the victim process.

Let us consider option 1. Whenever an edge is either inserted in or removed from a local graph, the local site must also send a message to the coordinator to notify it of this modification. On receiving such a message, the coordinator updates its global graph.

Alternatively (option 2), a site can send a number of such changes in a single message periodically. Returning to our previous example, the coordinator process will maintain the global wait-for graph as depicted in Figure 18.4. When site S_2 inserts the edge $P_3 \rightarrow P_4$ in its local wait-for graph, it also sends a message to the coordinator. Similarly, when site S_1 deletes the edge $P_5 \rightarrow P_1$ because P_1 has released a resource that was requested by P_5, an appropriate message is sent to the coordinator.

Note that no matter which option is used, unnecessary rollbacks may occur, as a result of two situations:

1. **False cycles** may exist in the global wait-for graph. To illustrate this point, we consider a snapshot of the system as depicted in Figure 18.5. Suppose that P_2 releases the resource it is holding in site S_1, resulting in the deletion of the edge $P_1 \rightarrow P_2$ in site S_1. Process P_2 then requests a resource held by P_3 at site S_2, resulting in the addition of the edge $P_2 \rightarrow P_3$ in site S_2. If the *insert* $P_2 \rightarrow P_3$ message from site S_2 arrives before the *delete* $P_1 \rightarrow P_2$ message from site S_1, the coordinator may discover the false cycle $P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow P_1$ after the *insert* (but before the *delete*). Deadlock recovery may be initiated, although no deadlock has occurred.
2. Unnecessary rollbacks may also result when a deadlock has indeed occurred and a victim has been picked, but *at the same time* one of the processes has been aborted for reasons unrelated to the deadlock (as when a process has exceeded its allocated time). For example, suppose that site S_1 in Figure 18.3 decides to abort P_2. At the same time, the coordinator has discovered a cycle and picked P_3 as a victim. Both P_2 and P_3 are now rolled back, although only P_2 needed to be rolled back.

Let us now consider a centralized deadlock-detection algorithm using option 3 that detects all deadlocks that actually occur and does not detect false deadlocks. To avoid the report of false deadlocks, we require that requests from different sites be appended with unique identifiers (or timestamps). When

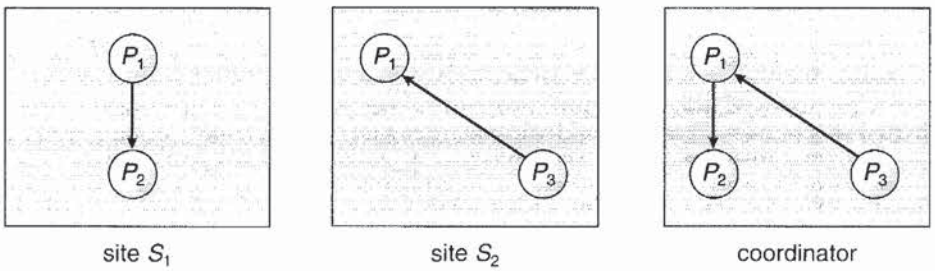


Figure 18.5 Local and global wait-for graphs.

process P_i, at site S_1, requests a resource from P_j, at site S_2, a request message with timestamp TS is sent. The edge $P_i \rightarrow P_j$ with the label TS is inserted in the local wait-for graph of S_1. This edge is inserted in the local wait-for graph of site S_2 only if site S_2 has received the request message and cannot immediately grant the requested resource. A request from P_i to P_j in the same site is handled in the usual manner; no timestamps are associated with the edge $P_i \rightarrow P_j$.

The detection algorithm is as follows:

1. The controller sends an initiating message to each site in the system.
2. On receiving this message, a site sends its local wait-for graph to the coordinator. Each of these wait-for graphs contains all the local information the site has about the state of the real graph. The graph reflects an instantaneous state of the site, but it is not synchronized with respect to any other site.
3. When the controller has received a reply from each site, it constructs a graph as follows:
   a. The constructed graph contains a vertex for every process in the system.
   b. The graph has an edge $P_i \rightarrow P_j$ *if and only if* there is an edge $P_i \rightarrow P_j$ in one of the wait-for graphs *or* an edge $P_i \rightarrow P_j$ with some label TS in more than one wait-for graph.

If the constructed graph contains a cycle, then the system is in a deadlocked state. If the constructed graph does not contain a cycle, then the system was not in a deadlocked state when the detection algorithm was invoked as result of the initiating messages sent by the coordinator (in step 1).

18.5.2.2 Fully Distributed Approach

In the **fully distributed deadlock-detection algorithm**, all controllers share equally the responsibility for detecting deadlock. Every site constructs a wait-for graph that represents a part of the total graph, depending on the dynamic behavior of the system. The idea is that, if a deadlock exists, a cycle will appear in at least one of the partial graphs. We present one such algorithm, which involves construction of partial graphs in every site.

Each site maintains its own local wait-for graph. A local wait-for graph in this scheme differs from the one described earlier in that we add one additional node P_{ex} to the graph. An arc $P_i \rightarrow P_{ex}$ exists in the graph if P_i is waiting for a data item in another site being held by *any* process. Similarly, an arc $P_{ex} \rightarrow P_j$ exists in the graph if a process at another site is waiting to acquire a resource currently being held by P_j in this local site.

To illustrate this situation, we consider again the two local wait-for graphs of Figure 18.3. The addition of the node P_{ex} in both graphs results in the local wait-for graphs shown in Figure 18.6.

If a local wait-for graph contains a cycle that does not involve node P_{ex}, then the system is in a deadlocked state. If, however, a local graph contains a cycle involving P_{ex}, then this implies the *possibility* of a deadlock. To ascertain whether a deadlock does exist, we must invoke a distributed deadlock-detection algorithm.

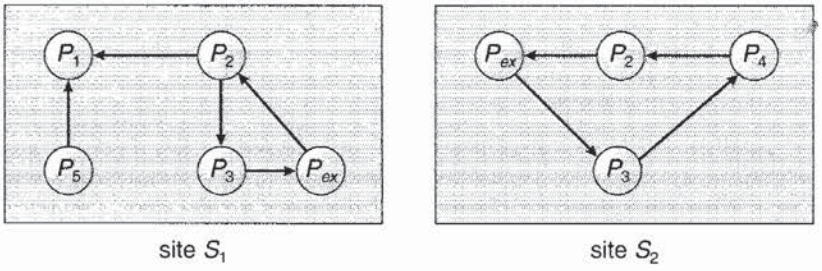


Figure 18.6 Augmented local wait-for graphs of Figure 18.3.

Suppose that, at site S_i, the local wait-for graph contains a cycle involving node P_{ex}. This cycle must be of the form

$$P_{ex} \rightarrow P_{k_1} \rightarrow P_{k_2} \rightarrow \ldots \rightarrow P_{k_n} \rightarrow P_{ex},$$

which indicates that process P_{k_n} in site S_i is waiting to acquire a data item located in some other site—say, S_j. On discovering this cycle, site S_i sends to site S_j a deadlock-detection message containing information about that cycle.

When site S_j receives this deadlock-detection message, it updates its local wait-for graph with the new information. Then it searches the newly constructed wait-for graph for a cycle not involving P_{ex}. If one exists, a deadlock is found, and an appropriate recovery scheme is invoked. If a cycle involving P_{ex} is discovered, then S_j transmits a deadlock-detection message to the appropriate site—say, S_k. Site S_k, in return, repeats the procedure. Thus, after a finite number of rounds, either a deadlock is discovered or the deadlock-detection computation halts.

To illustrate this procedure, we consider the local wait-for graphs of Figure 18.6. Suppose that site S_1 discovers the cycle

$$P_{ex} \rightarrow P_2 \rightarrow P_3 \rightarrow P_{ex}.$$

Since P_3 is waiting to acquire a data item in site S_2, a deadlock-detection message describing that cycle is transmitted from site S_1 to site S_2. When site S_2 receives this message, it updates its local wait-for graph, obtaining the wait-for graph of Figure 18.7. This graph contains the cycle

$$P_2 \rightarrow P_3 \rightarrow P_4 \rightarrow P_2,$$

which does not include node P_{ex}. Therefore, the system is in a deadlocked state, and an appropriate recovery scheme must be invoked.

Note that the outcome would be the same if site S_2 discovered the cycle first in its local wait-for graph and sent the deadlock-detection message to site S_1. In the worst case, both sites will discover the cycle at about the same time, and two deadlock-detection messages will be sent: one by S_1 to S_2 and another by S_2 to S_1. This situation results in unnecessary message transfer and overhead in updating the two local wait-for graphs and searching for cycles in both graphs.

To reduce message traffic, we assign to each process P_i a unique identifier, which we denote $ID(P_i)$. When site S_k discovers that its local wait-for graph contains a cycle involving node P_{ex} of the form

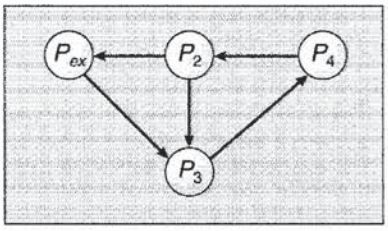


Figure 18.7 Augmented local wait-for graph in site S_2 of Figure 18.6.

$$P_{ex} \rightarrow P_{K_1} \rightarrow P_{K_2} \rightarrow \ldots \rightarrow P_{K_n} \rightarrow P_{ex},$$

it sends a deadlock-detection message to another site only if

$$ID(P_{K_n}) < ID(P_{K_1}).$$

Otherwise, site S_k continues its normal execution, leaving the burden of initiating the deadlock-detection algorithm to some other site.

To illustrate this scheme, we consider again the wait-for graphs maintained at sites S_1 and S_2 of Figure 18.6. Suppose that

$$ID(P_1) < ID(P_2) < ID(P_3) < ID(P_4).$$

Let both sites discover these local cycles at about the same time. The cycle in site S_1 is of the form

$$P_{ex} \rightarrow P_2 \rightarrow P_3 \rightarrow P_{ex}.$$

Since $ID(P_3) > ID(P_2)$, site S_1 does not send a deadlock-detection message to site S_2.

The cycle in site S_2 is of the form

$$P_{ex} \rightarrow P_3 \rightarrow P_4 \rightarrow P_2 \rightarrow P_{ex}.$$

Since $ID(P_2) < ID(P_3)$, site S_2 does send a deadlock-detection message to site S_1, which, on receiving the message, updates its local wait-for graph. Site S_1 then searches for a cycle in the graph and discovers that the system is in a deadlocked state.

18.6 Election Algorithms

As we pointed out in Section 18.3, many distributed algorithms employ a coordinator process that performs functions needed by the other processes in the system. These functions include enforcing mutual exclusion, maintaining a global wait-for graph for deadlock detection, replacing a lost token, and controlling an input or output device in the system. If the coordinator process fails due to the failure of the site at which it resides, the system can continue

execution only by restarting a new copy of the coordinator on some other site. The algorithms that determine where a new copy of the coordinator should be restarted are called **election algorithms**.

Election algorithms assume that a unique priority number is associated with each active process in the system. For ease of notation, we assume that the priority number of process P_i is i. To simplify our discussion, we assume a one-to-one correspondence between processes and sites and thus refer to both as processes. The coordinator is always the process with the largest priority number. Hence, when a coordinator fails, the algorithm must elect that active process with the largest priority number. This number must be sent to each active process in the system. In addition, the algorithm must provide a mechanism for a recovered process to identify the current coordinator.

In this section, we present examples of election algorithms for two different configurations of distributed systems. The first algorithm applies to systems where every process can send a message to every other process in the system. The second algorithm applies to systems organized as a ring (logically or physically). Both algorithms require n^2 messages for an election, where n is the number of processes in the system. We assume that a process that has failed knows on recovery that it has indeed failed and thus takes appropriate actions to rejoin the set of active processes.

18.6.1 The Bully Algorithm

Suppose that process P_i sends a request that is not answered by the coordinator within a time interval T. In this situation, it is assumed that the coordinator has failed, and P_i tries to elect itself as the new coordinator. This task is completed through the following algorithm.

Process P_i sends an election message to every process with a higher priority number. Process P_i then waits for a time interval T for an answer from any one of these processes.

If no response is received within time T, P_i assumes that all processes with numbers greater than i have failed and elects itself the new coordinator. Process P_i restarts a new copy of the coordinator and sends a message to inform all active processes with priority numbers less than i that P_i is the new coordinator.

However, if an answer is received, P_i begins a time interval T', waiting to receive a message informing it that a process with a higher priority number has been elected. (That is, some other process is electing itself coordinator and should report the results within time T'.) If no message is sent within T', then the process with a higher number is assumed to have failed, and process P_i should restart the algorithm.

If P_i is not the coordinator, then, at any time during execution, P_i may receive one of the following two messages from process P_j:

1. P_j is the new coordinator ($j > i$). Process P_i, in turn, records this information.
2. P_j has started an election ($j < i$). Process P_i sends a response to P_j and begins its own election algorithm, provided that P_i has not already initiated such an election.

The process that completes its algorithm has the highest number and is elected as the coordinator. It has sent its number to all active processes with smaller

numbers. After a failed process recovers, it immediately begins execution of the same algorithm. If there are no active processes with higher numbers, the recovered process forces all processes with lower numbers to let it become the coordinator process, even if there is a currently active coordinator with a lower number. For this reason, the algorithm is termed the **bully algorithm**.

We can demonstrate the operation of the algorithm with a simple example of a system consisting of processes P_1 through P_4. The operations are as follows:

1. All processes are active; P_4 is the coordinator process.
2. P_1 and P_4 fail. P_2 determines that P_4 has failed by sending a request that is not answered within time T. P_2 then begins its election algorithm by sending a request to P_3.
3. P_3 receives the request, responds to P_2, and begins its own algorithm by sending an election request to P_4.
4. P_2 receives P_3's response and begins waiting for an interval T'.
5. P_4 does not respond within an interval T, so P_3 elects itself the new coordinator and sends the number 3 to P_2 and P_1. (P_1 does not receive the number, since it has failed.)
6. Later, when P_1 recovers, it sends an election request to P_2, P_3, and P_4.
7. P_2 and P_3 respond to P_1 and begin their own election algorithms. P_3 will again be elected, through the same events as before.
8. Finally, P_4 recovers and notifies P_1, P_2, and P_3 that it is the current coordinator. (P_4 sends no election requests, since it is the process with the highest number in the system.)

18.6.2 The Ring Algorithm

The **ring algorithm** assumes that the links are unidirectional and that each process sends its messages to the neighbor on the right. The main data structure used by the algorithm is the **active list**, a list that contains the priority numbers of all active processes in the system when the algorithm ends; each process maintains its own active list. The algorithm works as follows:

1. If process P_i detects a coordinator failure, it creates a new active list that is initially empty. It then sends a message *elect*(i) to its right neighbor and adds the number i to its active list.
2. If P_i receives a message *elect*(j) from the process on the left, it must respond in one of three ways:
   a. If this is the first *elect* message it has seen or sent, P_i creates a new active list with the numbers i and j. It then sends the message *elect*(i), followed by the message *elect*(j).
   b. If $i \neq j$—that is, the message received does not contain P_i's number—then P_i adds j to its active list and forwards the message to its right neighbor.

c. If $i = j$—that is, P_i receives the message *elect*(i)—then the active list for P_i now contains the numbers of all the active processes in the system. Process P_i can now determine the largest number in the active list to identify the new coordinator process.

This algorithm does not specify how a recovering process determines the number of the current coordinator process. One solution requires a recovering process to send an inquiry message. This message is forwarded around the ring to the current coordinator, which in turn sends a reply containing its number.

18.7 Reaching Agreement

For a system to be reliable, we need a mechanism that allows a set of processes to agree on a common *value*. Such an agreement may not take place, for several reasons. First, the communication medium may be faulty, resulting in lost or garbled messages. Second, the processes themselves may be faulty, resulting in unpredictable process behavior. The best we can hope for in this case is that processes fail in a clean way, stopping their execution without deviating from their normal execution pattern. In the worst case, processes may send garbled or incorrect messages to other processes or even collaborate with other failed processes in an attempt to destroy the integrity of the system.

The **Byzantine generals problem** provides an analogy for this situation. Several divisions of the Byzantine army, each commanded by its own general, surround an enemy camp. The Byzantine generals must reach agreement on whether or not to attack the enemy at dawn. It is crucial that all generals agree, since an attack by only some of the divisions would result in defeat. The various divisions are geographically dispersed, and the generals can communicate with one another only via messengers who run from camp to camp. The generals may not be able to reach agreement for at least two major reasons:

1. Messengers may get caught by the enemy and thus may be unable to deliver their messages. This situation corresponds to unreliable communication in a computer system and is discussed further in Section 18.7.1.
2. Generals may be *traitors*, trying to prevent the *loyal* generals from reaching an agreement. This situation corresponds to faulty processes in a computer system and is discussed further in Section 18.7.2.

18.7.1 Unreliable Communications

Let us first assume that, if processes fail, they do so in a clean way and that the communication medium is unreliable. Suppose that process P_i at site S_1, which has sent a message to process P_j at site S_2, needs to know whether P_j has received the message so that it can decide how to proceed with its computation. For example, P_i may decide to compute a function *foo* if P_j has received its message or to compute a function *boo* if P_j has not received the message (because of some hardware failure).

To detect failures, we can use a **time-out scheme** similar to the one described in Section 16.7.1. When P_i sends out a message, it also specifies

a time interval during which it is willing to wait for an acknowledgment message from P_j. When P_j receives the message, it immediately sends an acknowledgment to P_i. If P_i receives the acknowledgment message within the specified time interval, it can safely conclude that P_j has received its message. If, however, a time-out occurs, then P_i needs to retransmit its message and wait for an acknowledgment. This procedure continues until P_i either gets the acknowledgment message back or is notified by the system that site S_2 is down. In the first case, it will compute S; in the latter case, it will compute F. Note that, if these are the only two viable alternatives, P_i must wait until it has been notified that one of the situations has occurred.

Suppose now that P_j also needs to know that P_i has received its acknowledgment message, so that it can decide how to proceed with its computation. For example, P_j may want to compute *foo* only if it is assured that P_i got its acknowledgment. In other words, P_i and P_j will compute *foo* if and only if both have agreed on it. It turns out that, in the presence of failure, it is not possible to accomplish this task. More precisely, it is not possible in a distributed environment for processes P_i and P_j to agree completely on their respective states.

To prove this claim, let us suppose that a minimal sequence of message transfers exists such that, after the messages have been delivered, both processes agree to compute *foo*. Let m' be the last message sent by P_i to P_j. Since P_i does not know whether its message will arrive at P_j (since the message may be lost due to a failure), P_i will execute *foo* regardless of the outcome of the message delivery. Thus, m' could be removed from the sequence without affecting the decision procedure. Hence, the original sequence was not minimal, contradicting our assumption and showing that there is no sequence. The processes can never be sure that both will compute *foo*.

18.7.2 Faulty Processes

Now let us assume that the communication medium is reliable but that processes can fail in unpredictable ways. Consider a system of n processes, of which no more than m are faulty. Suppose that each process P_i has some private value of V_i. We wish to devise an algorithm that allows each nonfaulty process P_i to construct a vector $X_i = (A_{i,1}, A_{i,2}, ..., A_{i,n})$ such that the following conditions exist:

1. If P_j is a nonfaulty process, then $A_{i,j} = V_j$.
2. If P_i and P_j are both nonfaulty processes, then $X_i = X_j$.

There are many solutions to this problem, and they share the following properties:

1. A correct algorithm can be devised only if $n \geq 3 \times m + 1$.
2. The worst-case delay for reaching agreement is proportionate to $m + 1$ message-passing delays.
3. The number of messages required for reaching agreement is large. No single process is trustworthy, so all processes must collect all information and make their own decisions.

Rather than presenting a general solution, which would be complicated, we present an algorithm for the simple case where $m = 1$ and $n = 4$. The algorithm requires two rounds of information exchange:

1. Each process sends its private value to the other three processes.
2. Each process sends the information it has obtained in the first round to all other processes.

A faulty process obviously may refuse to send messages. In this case, a nonfaulty process can choose an arbitrary value and pretend that the value was sent by the faulty process.

Once these two rounds are completed, a nonfaulty process P_i can construct its vector $X_i = (A_{i,1}, A_{i,2}, A_{i,3}, A_{i,4})$ as follows:

1. $A_{i,i} = V_i$.
2. For $j \neq i$, if at least two of the three values reported for process P_j (in the two rounds of exchange) agree, then the majority value is used to set the value of $A_{i,j}$. Otherwise, a default value—say, *nil*—is used to set the value of $A_{i,j}$.

18.8 Summary

In a distributed system with no common memory and no common clock, it is sometimes impossible to determine the exact order in which two events occur. The *happened-before* relation is only a partial ordering of the events in a distributed system. Timestamps can be used to provide a consistent event ordering.

Mutual exclusion in a distributed environment can be implemented in a variety of ways. In a centralized approach, one of the processes in the system is chosen to coordinate the entry to the critical section. In the fully distributed approach, the decision making is distributed across the entire system. A distributed algorithm, which is applicable to ring-structured networks, is the token-passing approach.

For atomicity to be ensured, all the sites in which a transaction T has executed must agree on the final outcome of the execution. T either commits at all sites or aborts at all sites. To ensure this property, the transaction coordinator of T must execute a commit protocol. The most widely used commit protocol is the 2PC protocol.

The various concurrency-control schemes that can be used in a centralized system can be modified for use in a distributed environment. In the case of locking protocols, we need only change the way the lock manager is implemented. In the case of timestamping and validation schemes, the only change needed is the development of a mechanism for generating unique global timestamps. The mechanism can either concatenate a local timestamp with the site identification or advance local clocks whenever a message arrives that has a larger timestamp.

The primary method for dealing with deadlocks in a distributed environment is deadlock detection. The main problem is deciding how to maintain the

wait-for graph. Methods for organizing the wait-for graph include a centralized approach and a fully distributed approach.

Some distributed algorithms require the use of a coordinator. If the coordinator fails because of the failure of the site at which it resides, the system can continue execution only by restarting a new copy of the coordinator on some other site. It can do so by maintaining a backup coordinator that is ready to assume responsibility if the coordinator fails. Another approach is to choose the new coordinator after the coordinator has failed. The algorithms that determine where a new copy of the coordinator should be restarted are called election algorithms. Two algorithms, the bully algorithm and the ring algorithm, can be used to elect a new coordinator in case of failures.

Exercises

18.1 Discuss the advantages and disadvantages of the two methods we presented for generating globally unique timestamps.

18.2 The logical clock timestamp scheme presented in this chapter provides the following guarantee: If event A happens before event B, then the timestamp of A is less than the timestamp of B. Note, however, that one cannot order two events based only on their timestamps. The fact that an event C has a timestamp that is less than the timestamp of event D does not necessarily mean that event C happened before event D; C and D could be concurrent events in the system. Discuss ways in which the logical clock timestamp scheme could be extended to distinguish concurrent events from events that can be ordered by the *happens-before* relationship.

18.3 Your company is building a computer network, and you are asked to write an algorithm for achieving distributed mutual exclusion. Which scheme will you use? Explain your choice.

18.4 Why is deadlock detection much more expensive in a distributed environment than in a centralized environment?

18.5 Your company is building a computer network, and you are asked to develop a scheme for dealing with the deadlock problem.

a. Would you use a deadlock-detection scheme or a deadlock-prevention scheme?

b. If you were to use a deadlock-prevention scheme, which one would you use? Explain your choice.

c. If you were to use a deadlock-detection scheme, which one would you use? Explain your choice.

18.6 Under what circumstances does the wait–die scheme perform better than the wound–wait scheme for granting resources to concurrently executing transactions?

18.7 Consider the centralized and the fully distributed approaches to deadlock detection. Compare the two algorithms in terms of message complexity.

18.8 Consider the following *hierarchical* deadlock-detection algorithm, in which the global wait-for graph is distributed over a number of different *controllers*, which are organized in a tree. Each non-leaf controller maintains a wait-for graph that contains relevant information from the graphs of the controllers in the subtree below it. In particular, let S_A, S_B, and S_C be controllers such that S_C is the lowest common ancestor of S_A and S_B (S_C must be unique, since we are dealing with a tree). Suppose that node T_i appears in the local wait-for graph of controllers S_A and S_B. Then T_i must also appear in the local wait-for graph of

- Controller S_C
- Every controller in the path from S_C to S_A
- Every controller in the path from S_C to S_B

In addition, if T_i and T_j appear in the wait-for graph of controller S_D and there exists a path from T_i to T_j in the wait-for graph of one of the children of S_D, then an edge $T_i \rightarrow T_j$ must be in the wait-for graph of S_D.

Show that, if a cycle exists in any of the wait-for graphs, then the system is deadlocked.

18.9 Derive an election algorithm for bidirectional rings that is more efficient than the one presented in this chapter. How many messages are needed for n processes?

18.10 Consider a setting where processors are not associated with unique identifiers but the total number of processors is known and the processors are organized along a bidirectional ring. Is it possible to derive an election algorithm for such a setting?

18.11 Consider a failure that occurs during 2PC for a transaction. For each possible failure, explain how 2PC ensures transaction atomicity despite the failure.

18.12 Consider the following failure model for faulty processors. Processors follow the protocol but might fail at unexpected points in time. When processors fail, they simply stop functioning and do not continue to participate in the distributed system. Given such a failure model, design an algorithm for reaching agreement among a set of processors. Discuss the conditions under which agreement could be reached.

Bibliographical Notes

The distributed algorithm for extending the *happened-before* relation to a consistent total ordering of all the events in the system was developed by Lamport [1978b]. Further discussions of using logical time to characterize the behavior of distributed systems can be found in Fidge [1991], Raynal and Singhal [1996], Babaoglu and Marzullo [1993], Schwarz and Mattern [1994], and Mattern [1988].

The first general algorithm for implementing mutual exclusion in a distributed environment was also developed by Lamport [1978b]. Lamport's scheme requires $3 \times (n - 1)$ messages per critical-section entry. Subsequently, Ricart and Agrawala [1981] proposed a distributed algorithm that requires only $2 \times (n - 1)$ messages. Their algorithm is presented in Section 18.2.2. A square-root algorithm for distributed mutual exclusion was described by Maekawa [1985]. The token-passing algorithm for ring-structured systems presented in Section 18.2.3 was developed by Lann [1977]. Carvalho and Roucairol [1983] discussed mutual exclusion in computer networks, and Agrawal and Abbadi [1991] described an efficient and fault-tolerant solution of distributed mutual exclusion. A simple taxonomy for distributed mutual-exclusion algorithms was presented by Raynal [1991].

The issue of distributed synchronization was discussed by Reed and Kanodia [1979] (shared-memory environment), Lamport [1978b], Lamport [1978a], and Schneider [1982] (totally disjoint processes). A distributed solution to the dining-philosophers problem was presented by Chang [1980].

The 2PC protocol was developed by Lampson and Sturgis [1976] and Gray [1978]. Mohan and Lindsay [1983] discussed two modified versions of 2PC, called presume commit and presume abort, that reduce the overhead of 2PC by defining default assumptions regarding the fate of transactions.

Papers dealing with the problems of implementing the transaction concept in a distributed database were presented by Gray [1981], Traiger et al. [1982], and Spector and Schwarz [1983]. Comprehensive discussions of distributed concurrency control were offered by Bernstein et al. [1987]. Rosenkrantz et al. [1978] reported the timestamp distributed deadlock-prevention algorithm. The fully distributed deadlock-detection scheme presented in Section 18.5.2 was developed by Obermarck [1982]. The hierarchical deadlock-detection scheme of Exercise 18.4 appeared in Menasce and Muntz [1979]. Knapp [1987] and Singhal [1989] offered surveys of deadlock detection in distributed systems. Deadlocks can also be detected by taking global snapshots of a distributed system, as discussed in Chandy and Lamport [1985].

The Byzantine generals problem was discussed by Lamport et al. [1982] and Pease et al. [1980]. The bully algorithm was presented by Garcia-Molina [1982], and the election algorithm for a ring-structured system was written by Lann [1977].

Part Seven

Special-Purpose Systems

Our coverage of operating-system issues thus far has focused mainly on general-purpose computing systems. There are, however, special-purpose systems with requirements different from those of many of the systems we have described.

A *real-time system* is a computer system that requires not only that computed results be "correct" but also that the results be produced within a specified deadline period. Results produced after the deadline has passed—even if correct—may be of no real value. For such systems, many traditional operating-system scheduling algorithms must be modified to meet the stringent timing deadlines.

A *multimedia system* must be able to handle not only conventional data, such as text files, programs, and word-processing documents, but also multimedia data. Multimedia data consist of continuous-media data (audio and video) as well as conventional data. Continuous-media data—such as frames of video—must be delivered according to certain time restrictions (for example, 30 frames per second). The demands of handling continuous-media data require significant changes in operating-system structure, most notably in memory, disk, and network management.



Real-Time Systems



Our coverage of operating-system issues thus far has focused mainly on general-purpose computing systems (for example, desktop and server systems). In this chapter, we turn our attention to real-time computing systems. The requirements of real-time systems differ from those of many of the systems we have described, largely because real-time systems must produce results within certain deadlines. In this chapter we provide an overview of real-time computer systems and describe how real-time operating systems must be constructed to meet the stringent timing requirements of these systems.

CHAPTER OBJECTIVES

- To explain the timing requirements of real-time systems.
- To distinguish between hard and soft real-time systems.
- To discuss the defining characteristics of real-time systems.
- To describe scheduling algorithms for hard real-time systems.

19.1 Overview

A **real-time system** is a computer system that requires not only that the computing results be "correct" but also that the results be produced within a specified deadline period. Results produced after the deadline has passed—even if correct—may be of no real value. To illustrate, consider an autonomous robot that delivers mail in an office complex. If its vision-control system identifies a wall *after* the robot has walked into it, despite correctly identifying the wall, the system has not met its requirement. Contrast this timing requirement with the much less strict demands of other systems. In an interactive desktop computer system, it is desirable to provide a quick response time to the interactive user, but it is not mandatory to do so. Some systems — such as a batch-processing system— may have no timing requirements whatsoever.

Real-time systems executing on traditional computer hardware are used in a wide range of applications. In addition, many real-time systems are

embedded in "specialized devices," such as ordinary home appliances (for example, microwave ovens and dishwashers), consumer digital devices (for example, cameras and MP3 players), and communication devices (for example, cellular telephones and Blackberry handheld devices). They are also present in larger entities, such as automobiles and airplanes. An **embedded system** is a computing device that is part of a larger system in which the presence of a computing device is often not obvious to the user.

To illustrate, consider an embedded system for controlling a home dishwasher. The embedded system may allow various options for scheduling the operation of the dishwasher—the water temperature, the type of cleaning (light or heavy), even a timer indicating when the dishwasher is to start. Most likely, the user of the dishwasher is unaware that there is in fact a computer embedded in the appliance. As another example, consider an embedded system controlling antilock brakes in an automobile. Each wheel in the automobile has a sensor detecting how much sliding and traction are occurring, and each sensor continually sends its data to the system controller. Taking the results from these sensors, the controller tells the braking mechanism in each wheel how much braking pressure to apply. Again, to the user (in this instance, the driver of the automobile), the presence of an embedded computer system may not be apparent. It is important to note, however, that not all embedded systems are real-time. For example, an embedded system controlling a home furnace may have no real-time requirements whatsoever.

Some real-time systems are identified as **safety-critical systems**. In a safety-critical system, incorrect operation—usually due to a missed deadline—results in some sort of "catastrophe." Examples of safety-critical systems include weapons systems, antilock brake systems, flight-management systems, and health-related embedded systems, such as pacemakers. In these scenarios, the real-time system *must* respond to events by the specified deadlines; otherwise, serious injury—or worse—might occur. However, a significant majority of embedded systems do not qualify as safety-critical, including FAX machines, microwave ovens, wristwatches, and networking devices such as switches and routers. For these devices, missing deadline requirements results in nothing more than perhaps an unhappy user.

Real-time computing is of two types: hard and soft. A **hard real-time system** has the most stringent requirements, guaranteeing that critical real-time tasks be completed within their deadlines. Safety-critical systems are typically hard real-time systems. A **soft real-time system** is less restrictive, simply providing that a critical real-time task will receive priority over other tasks and that it will retain that priority until it completes. Many commercial operating systems—as well as Linux—provide soft real-time support.

19.2 System Characteristics

In this section, we explore the characteristics of real-time systems and address issues related to designing both soft and hard real-time operating systems.

The following characteristics are typical of many real-time systems:

- Single purpose
- Small size

- Inexpensively mass-produced
- Specific timing requirements

We next examine each of these characteristics.

Unlike PCs, which are put to many uses, a real-time system typically serves only a single purpose, such as controlling antilock brakes or delivering music on an MP3 player. It is unlikely that a real-time system controlling an airliner's navigation system will also play DVDs! The design of a real-time operating system reflects its single-purpose nature and is often quite simple.

Many real-time systems exist in environments where physical space is constrained. Consider the amount of space available in a wristwatch or a microwave oven—it is considerably less than what is available in a desktop computer. As a result of space constraints, most real-time systems lack both the CPU processing power and the amount of memory available in standard desktop PCs. Whereas most contemporary desktop and server systems use 32- or 64-bit processors, many real-time systems run on 8- or 16-bit processors. Similarly, a desktop PC might have several gigabytes of physical memory, whereas a real-time system might have less than a megabyte. We refer to the **footprint** of a system as the amount of memory required to run the operating system and its applications. Because the amount of memory is limited, most real-time operating systems must have small footprints.

Next, consider where many real-time systems are implemented: They are often found in home appliances and consumer devices. Devices such as digital cameras, microwave ovens, and thermostats are mass-produced in very cost-conscious environments. Thus, the microprocessors for real-time systems must also be inexpensively mass-produced.

One technique for reducing the cost of an embedded controller is to use an alternative technique for organizing the components of the computer system. Rather than organizing the computer around the structure shown in Figure 19.1, where buses provide the interconnection mechanism to individual components, many embedded system controllers use a strategy known as **system-on-chip (SOC)**. Here, the CPU, memory (including cache), memory-

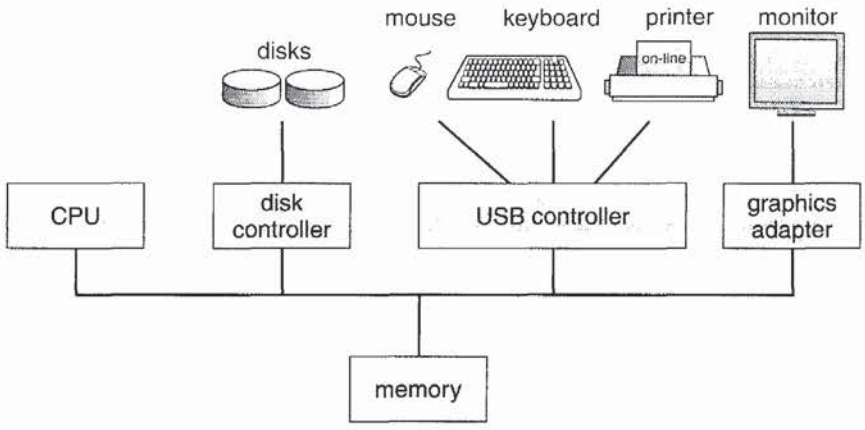


Figure 19.1 Bus-oriented organization.

management-unit (MMU), and any attached peripheral ports, such as USB ports, are contained in a single integrated circuit. The SOC strategy is typically less expensive than the bus-oriented organization of Figure 19.1.

We turn now to the final characteristic identified above for real-time systems: specific timing requirements. It is, in fact, the defining characteristic of such systems. Accordingly, the defining characteristic of both hard and soft real-time operating systems is to support the timing requirements of real-time tasks, and the remainder of this chapter focuses on this issue. Real-time operating systems meet timing requirements by using scheduling algorithms that give real-time processes the highest scheduling priorities. Furthermore, schedulers must ensure that the priority of a real-time task does not degrade over time. A second, somewhat related, technique for addressing timing requirements is by minimizing the response time to events such as interrupts.

19.3 Features of Real-Time Kernels

In this section, we discuss the features necessary for designing an operating system that supports real-time processes. Before we begin, though, let's consider what is typically *not* needed for a real-time system. We begin by examining several features provided in many of the operating systems discussed so far in this text, including Linux, UNIX, and the various versions of Windows. These systems typically provide support for the following:

- A variety of peripheral devices such as graphical displays, CD, and DVD drives
- Protection and security mechanisms
- Multiple users

Supporting these features often results in a sophisticated—and large—kernel. For example, Windows XP has over forty million lines of source code. In contrast, a typical real-time operating system usually has a very simple design, often written in thousands rather than millions of lines of source code. We would not expect these simple systems to include the features listed above.

But why don't real-time systems provide these features, which are crucial to standard desktop and server systems? There are several reasons, but three are most prominent. First, because most real-time systems serve a single purpose, they simply do not require many of the features found in a desktop PC. Consider a digital wristwatch: It obviously has no need to support a disk drive or DVD, let alone virtual memory. Furthermore, a typical real-time system does not include the notion of a user: The system simply supports a small number of tasks, which often await input from hardware devices (sensors, vision identification, and so forth). Second, the features supported by standard desktop operating systems are impossible to provide without fast processors and large amounts of memory. Both of these are unavailable in real-time systems due to space constraints, as explained earlier. In addition, many real-time systems lack sufficient space to support peripheral disk drives or graphical displays, although some systems may support file systems using nonvolatile memory (NVRAM). Third, supporting features common in standard

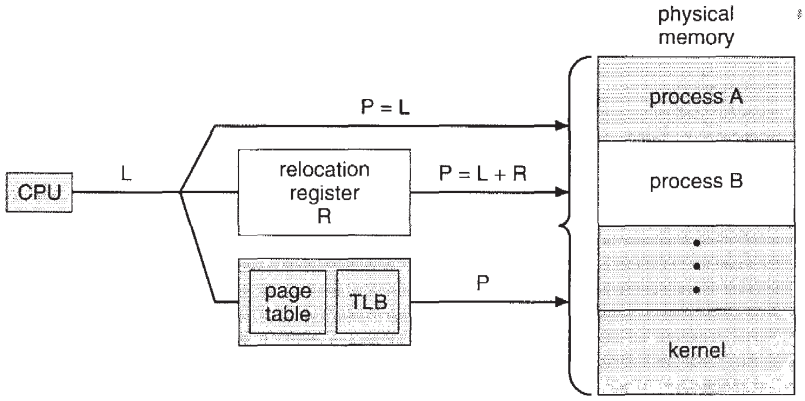


Figure 19.2 Address translation in real-time systems.

desktop computing environments would greatly increase the cost of real-time systems, which could make such systems economically impractical.

Additional considerations apply when considering virtual memory in a real-time system. Providing virtual memory features as described in Chapter 9 require the system include a memory management unit (MMU) for translating logical to physical addresses. However, MMUs typically increase the cost and power consumption of the system. In addition, the time required to translate logical addresses to physical addresses—especially in the case of a translation look-aside buffer (TLB) miss—may be prohibitive in a hard real-time environment. In the following we examine several appraoches for translating addresses in real-time systems.

Figure 19.2 illustrates three different strategies for managing address translation available to designers of real-time operating systems. In this scenario, the CPU generates logical address L that must be mapped to physical address P. The first approach is to bypass logical addresses and have the CPU generate physical addresses directly. This technique—known as **real-addressing mode**—does not employ virtual memory techniques and is effectively stating that P equals L. One problem with real-addressing mode is the absence of memory protection between processes. Real-addressing mode may also require that programmers specify the physical location where their programs are loaded into memory. However, the benefit of this approach is that the system is quite fast, as no time is spent on address translation. Real-addressing mode is quite common in embedded systems with hard real-time constraints. In fact, some real-time operating systems running on microprocessors containing an MMU actually disable the MMU to gain the performance benefit of referencing physical addresses directly.

A second strategy for translating addresses is to use an approach similar to the dynamic relocation register shown in Figure 8.4. In this scenario, a relocation register R is set to the memory location where a program is loaded. The physical address P is generated by adding the contents of the relocation register R to L. Some real-time systems configure the MMU to perform this way. The obvious benefit of this strategy is that the MMU can easily translate logical addresses to physical addresses using $P = L + R$. However, this system still suffers from a lack of memory protection between processes.

The last approach is for the real-time system to provide full virtual memory functionality as described in Chapter 9. In this instance, address translation takes place via page tables and a translation look-aside buffer, or TLB. In addition to allowing a program to be loaded at any memory location, this strategy also provides memory protection between processes. For systems without attached disk drives, demand paging and swapping may not be possible. However, systems may provide such features using NVRAM flash memory. The LynxOS and OnCore Systems are examples of real-time operating systems providing full support for virtual memory.

19.4 Implementing Real-Time Operating Systems

Keeping in mind the many possible variations, we now identify the features necessary for implementing a real-time operating system. This list is by no means absolute; some systems provide more features than we list below, while other systems provide fewer.

- Preemptive, priority-based scheduling
- Preemptive kernel
- Minimized latency

One notable feature we omit from this list is networking support. However, deciding whether to support networking protocols such as TCP/IP is simple: If the real-time system must be connected to a network, the operating system must provide networking capabilities. For example, a system that gathers real-time data and transmits it to a server must obviously include networking features. Alternatively, a self-contained embedded system requiring no interaction with other computer systems has no obvious networking requirement.

In the remainder of this section, we examine the basic requirements listed above and identify how they can be implemented in a real-time operating system.

19.4.1 Priority-Based Scheduling

The most important feature of a real-time operating system is to respond immediately to a real-time process as soon as that process requires the CPU. As a result, the scheduler for a real-time operating system must support a priority-based algorithm with preemption. Recall that priority-based scheduling algorithms assign each process a priority based on its importance; more important tasks are assigned higher priorities than those deemed less important. If the scheduler also supports preemption, a process currently running on the CPU will be preempted if a higher-priority process becomes available to run.

Preemptive, priority-based scheduling algorithms are discussed in detail in Chapter 5, where we also present examples of the soft real-time scheduling features of the Solaris, Windows XP, and Linux operating systems. Each of these systems assigns real-time processes the highest scheduling priority. For

example, Windows XP has 32 different priority levels; the highest levels—priority values 16 to 31—are reserved for real-time processes. Solaris and Linux have similar prioritization schemes.

Note, however, that providing a preemptive, priority-based scheduler only guarantees soft real-time functionality. Hard real-time systems must further guarantee that real-time tasks will be serviced in accord with their deadline requirements, and making such guarantees may require additional scheduling features. In Section 19.5, we cover scheduling algorithms appropriate for hard real-time systems.

19.4.2 Preemptive Kernels

Nonpreemptive kernels disallow preemption of a process running in kernel mode; a kernel-mode process will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU. In contrast, a preemptive kernel allows the preemption of a task running in kernel mode. Designing preemptive kernels can be quite difficult; and traditional user-oriented applications such as spreadsheets, word processors, and web browsers typically do not require such quick response times. As a result, some commercial desktop operating systems—such as Windows XP—are nonpreemptive.

However, to meet the timing requirements of real-time systems—in particular, hard real-time systems—preemptive kernels are mandatory. Otherwise, a real-time task might have to wait an arbitrarily long period of time while another task was active in the kernel.

There are various strategies for making a kernel preemptible. One approach is to insert **preemption points** in long-duration system calls. A preemption point checks to see whether a high-priority process needs to be run. If so, a context switch takes place. Then, when the high-priority process terminates, the interrupted process continues with the system call. Preemption points can be placed only at *safe* locations in the kernel—that is, only where kernel data structures are not being modified. A second strategy for making a kernel preemptible is through the use of synchronization mechanisms, which we discussed in Chapter 6. With this method, the kernel can always be preemptible, because any kernel data being updated are protected from modification by the high-priority process.

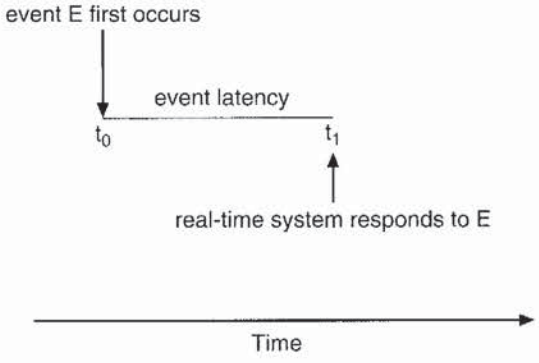


Figure 19.3 Event latency.

19.4.3 Minimizing Latency

Consider the event-driven nature of a real-time system: The system is typically waiting for an event in real time to occur. Events may arise either in software —as when a timer expires—or in hardware—as when a remote-controlled vehicle detects that it is approaching an obstruction. When an event occurs, the system must respond to and service it as quickly as possible. We refer to **event latency** as the amount of time that elapses from when an event occurs to when it is serviced (Figure 19.3).

Usually, different events have different latency requirements. For example, the latency requirement for an antilock brake system might be three to five milliseconds, meaning that from the time a wheel first detects that it is sliding, the system controlling the antilock brakes has three to five milliseconds to respond to and control the situation. Any response that takes longer might result in the automobile's veering out of control. In contrast, an embedded system controlling radar in an airliner might tolerate a latency period of several seconds.

Two types of latencies affect the performance of real-time systems:

1. Interrupt latency
2. Dispatch latency

Interrupt latency refers to the period of time from the arrival of an interrupt at the CPU to the start of the routine that services the interrupt. When an interrupt occurs, the operating system must first complete the instruction it is executing and determine the type of interrupt that occurred. It must then save the state of the current process before servicing the interrupt using the specific interrupt service routine (ISR). The total time required to perform these tasks is the interrupt latency (Figure 19.4). Obviously, it is crucial for real-time

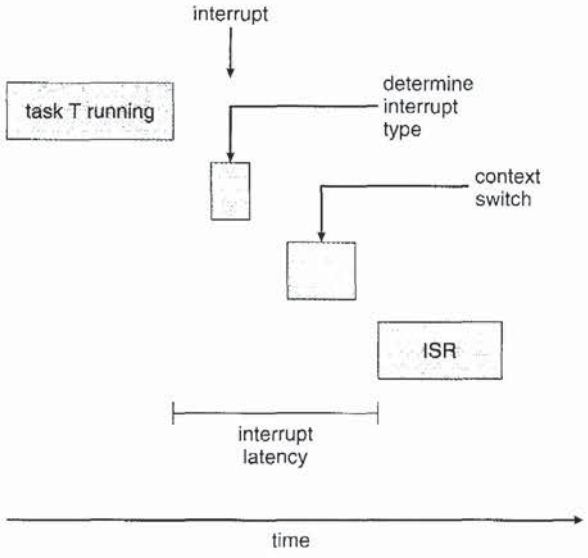


Figure 19.4 Interrupt latency.

operating systems to minimize interrupt latency to ensure that real-time tasks receive immediate attention.

One important factor contributing to interrupt latency is the amount of time interrupts may be disabled while kernel data structures are being updated. Real-time operating systems require that interrupts to be disabled for very short periods of time. However, for hard real-time systems, interrupt latency must not only be minimized, it must in fact be bounded to guarantee the deterministic behavior required of hard real-time kernels.

The amount of time required for the scheduling dispatcher to stop one process and start another is known as **dispatch latency**. Providing real-time tasks with immediate access to the CPU mandates that real-time operating systems minimize this latency. The most effective technique for keeping dispatch latency low is to provide preemptive kernels.

In Figure 19.5, we diagram the makeup of dispatch latency. The **conflict phase** of dispatch latency has two components:

1. Preemption of any process running in the kernel
2. Release by low-priority processes of resources needed by a high-priority process

As an example, in Solaris, the dispatch latency with preemption disabled is over 100 milliseconds. With preemption enabled, it is reduced to less than a millisecond.

One issue that can affect dispatch latency arises when a higher-priority process needs to read or modify kernel data that are currently being accessed by a lower-priority process—or a chain of lower-priority processes. As kernel

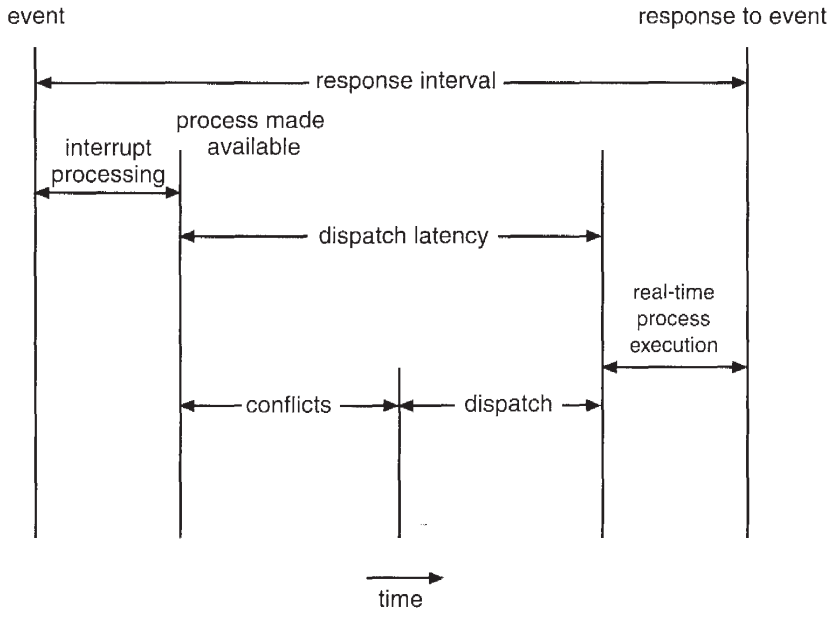


Figure 19.5 Dispatch latency.

data are typically protected with a lock, the higher-priority process will have to wait for a lower-priority one to finish with the resource. The situation becomes more complicated if the lower-priority process is preempted in favor of another process with a higher priority. As an example, assume we have three processes, L, M, and H, whose priorities follow the order $L < M < H$. Assume that process H requires resource R, which is currently being accessed by process L. Ordinarily, process H would wait for L to finish using resource R. However, now suppose that process M becomes runnable, thereby preempting process L. Indirectly, a process with a lower priority—process M—has affected how long process H must wait for L to relinquish resource R.

This problem, known as **priority inversion**, can be solved by use of the **priority-inheritance protocol**. According to this protocol, all processes that are accessing resources needed by a higher-priority process inherit the higher priority until they are finished with the resources in question. When they are finished, their priorities revert to their original values. In the example above, a priority-inheritance protocol allows process L to temporarily inherit the priority of process H, thereby preventing process M from preempting its execution. When process L has finished using resource R, it relinquishes its inherited priority from H and assumes its original priority. As resource R is now available, process H—not M—will run next.

19.5 Real-Time CPU Scheduling

Our coverage of scheduling so far has focused primarily on soft real-time systems. As mentioned, though, scheduling for such systems provides no guarantee on when a critical process will be scheduled; it guarantees only that the process will be given preference over noncritical processes. Hard real-time systems have stricter requirements. A task must be serviced by its deadline; service after the deadline has expired is the same as no service at all.

We now consider scheduling for hard real-time systems. Before we proceed with the details of the individual schedulers, however, we must define certain characteristics of the processes that are to be scheduled. First, the processes are considered **periodic**. That is, they require the CPU at constant intervals (periods). Each periodic process has a fixed processing time t once it acquires the CPU, a deadline d when it must be serviced by the CPU, and a period p. The relationship of the processing time, the deadline, and the period can be expressed as $0 \leq t \leq d \leq p$. The **rate** of a periodic task is $1/p$. Figure 19.6 illustrates the execution of a periodic process over time. Schedulers can take advantage of this relationship and assign priorities according to the deadline or rate requirements of a periodic process.

What is unusual about this form of scheduling is that a process may have to announce its deadline requirements to the scheduler. Then, using a technique known as an **admission-control** algorithm, the scheduler either admits the process, guaranteeing that the process will complete on time, or rejects the request as impossible if it cannot guarantee that the task will be serviced by its deadline.

In the following sections, we explore scheduling algorithms that address the deadline requirements of hard real-time systems.

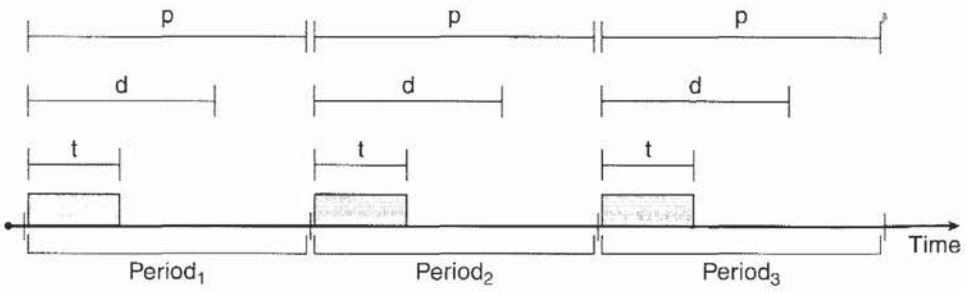


Figure 19.6 Periodic task.

19.5.1 Rate-Monotonic Scheduling

The **rate-monotonic** scheduling algorithm schedules periodic tasks using a static priority policy with preemption. If a lower-priority process is running and a higher-priority process becomes available to run, it will preempt the lower-priority process. Upon entering the system, each periodic task is assigned a priority inversely based on its period: The shorter the period, the higher the priority; the longer the period, the lower the priority. The rationale behind this policy is to assign a higher priority to tasks that require the CPU more often. Furthermore, rate-monotonic scheduling assumes that the processing time of a periodic process is the same for each CPU burst. That is, every time a process acquires the CPU, the duration of its CPU burst is the same.

Let's consider an example. We have two processes P_1 and P_2. The periods for P_1 and P_2 are 50 and 100, respectively—that is, $p_1 = 50$ and $p_2 = 100$. The processing times are $t_1 = 20$ for P_1 and $t_2 = 35$ for P_2. The deadline for each process requires that it complete its CPU burst by the start of its next period.

We must first ask ourselves whether it is possible to schedule these tasks so that each meets its deadlines. If we measure the CPU utilization of a process P_i as the ratio of its burst to its period—t_i/p_i—the CPU utilization of P_1 is $20/50 = 0.40$ and that of P_2 is $35/100 = 0.35$, for a total CPU utilization of 75 percent. Therefore, it seems we can schedule these tasks in such a way that both meet their deadlines and still leave the CPU with available cycles.

First, suppose we assign P_2 a higher priority than P_1. The execution of P_1 and P_2 is shown in Figure 19.7. As we can see, P_2 starts execution first and completes at time 35. At this point, P_1 starts; it completes its CPU burst at time 55. However, the first deadline for P_1 was at time 50, so the scheduler has caused P_1 to miss its deadline.

Now suppose we use rate-monotonic scheduling, in which we assign P_1 a higher priority than P_2, since the period of P_1 is shorter than that of P_2.

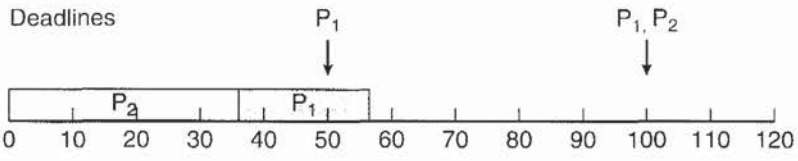


Figure 19.7 Scheduling of tasks when P_2 has a higher priority than P_1.

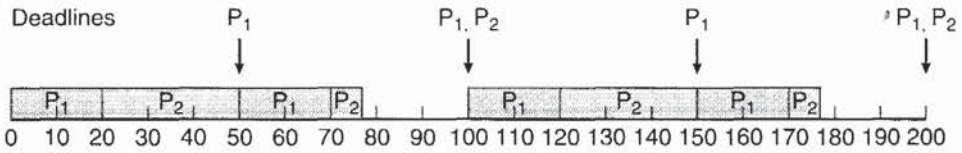


Figure 19.8 Rate-monotonic scheduling.

The execution of these processes is shown in Figure 19.8. P_1 starts first and completes its CPU burst at time 20, thereby meeting its first deadline. P_2 starts running at this point and runs until time 50. At this time, it is preempted by P_1, although it still has 5 milliseconds remaining in its CPU burst. P_1 completes its CPU burst at time 70, at which point the scheduler resumes P_2. P_2 completes its CPU burst at time 75, also meeting its first deadline. The system is idle until time 100, when P_1 is scheduled again.

Rate-monotonic scheduling is considered optimal in the sense that if a set of processes cannot be scheduled by this algorithm, it cannot be scheduled by any other algorithm that assigns static priorities. Let's next examine a set of processes that cannot be scheduled using the rate-monotonic algorithm. Assume that process P_1 has a period of $p_1 = 50$ and a CPU burst of $t_1 = 25$. For P_2, the corresponding values are $p_2 = 80$ and $t_2 = 35$. Rate-monotonic scheduling would assign process P_1 a higher priority, as it has the shorter period. The total CPU utilization of the two processes is $(25/50)+(35/80) = 0.94$, and it therefore seems logical that the two processes could be scheduled and still leave the CPU with 6 percent available time. The Gantt chart showing the scheduling of processes P_1 and P_2 is depicted in Figure 19.9. Initially, P_1 runs until it completes its CPU burst at time 25. Process P_2 then begins running and runs until time 50, when it is preempted by P_1. At this point, P_2 still has 10 milliseconds remaining in its CPU burst. Process P_1 runs until time 75; however, P_2 misses the deadline for completion of its CPU burst at time 80.

Despite being optimal, then, rate-monotonic scheduling has a limitation: CPU utilization is bounded, and it is not always possible to fully maximize CPU resources. The worst-case CPU utilization for scheduling N processes is

$$2(2^{1/n} - 1).$$

With one process in the system, CPU utilization is 100 percent; but it falls to approximately 69 percent as the number of processes approaches infinity. With two processes, CPU utilization is bounded at about 83 percent. Combined CPU utilization for the two processes scheduled in Figures 19.7 and 19.8 is 75 percent; and therefore, the rate-monotonic scheduling algorithm is guaranteed

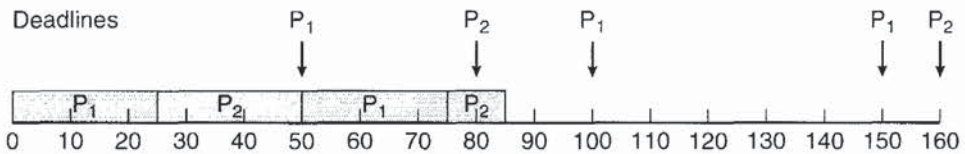


Figure 19.9 Missing deadlines with rate-monotonic scheduling.

to schedule them so that they can meet their deadlines. For the two processes scheduled in Figure 19.9, combined CPU utilization is approximately 94 percent; therefore, rate-monotonic scheduling cannot guarantee that they can be scheduled so that they meet their deadlines.

19.5.2 Earliest-Deadline-First Scheduling

Earliest-deadline-first (EDF) scheduling dynamically assigns priorities according to deadline. The earlier the deadline, the higher the priority; the later the deadline, the lower the priority. Under the EDF policy, when a process becomes runnable, it must announce its deadline requirements to the system. Priorities may have to be adjusted to reflect the deadline of the newly runnable process. Note how this differs from rate-monotonic scheduling, where priorities are fixed.

To illustrate EDF scheduling, we again schedule the processes shown in Figure 19.9, which failed to meet deadline requirements under rate-monotonic scheduling. Recall that P_1 has values of $p_1 = 50$ and $t_1 = 25$ and that P_2 has values $p_2 = 80$ and $t_2 = 35$. The EDF scheduling of these processes is shown in Figure 19.10. Process P_1 has the earliest deadline, so its initial priority is higher than that of process P_2. Process P_2 begins running at the end of the CPU burst for P_1. However, whereas rate-monotonic scheduling allows P_1 to preempt P_2 at the beginning of its next period at time 50, EDF scheduling allows process P_2 to continue running. P_2 now has a higher priority than P_1 because its next deadline (at time 80) is earlier than that of P_1 (at time 100). Thus, both P_1 and P_2 have met their first deadlines. Process P_1 again begins running at time 60 and completes its second CPU burst at time 85, also meeting its second deadline at time 100. P_2 begins running at this point, only to be preempted by P_1 at the start of its next period at time 100. P_2 is preempted because P_1 has an earlier deadline (time 150) than P_2 (time 160). At time 125, P_1 completes its CPU burst and P_2 resumes execution, finishing at time 145 and meeting its deadline as well. The system is idle until time 150, when P_1 is scheduled to run once again.

Unlike the rate-monotonic algorithm, EDF scheduling does not require that processes be periodic, nor must a process require a constant amount of CPU time per burst. The only requirement is that a process announce its deadline to the scheduler when it becomes runnable. The appeal of EDF scheduling is that it is theoretically optimal—theoretically, it can schedule processes so that each process can meet its deadline requirements and CPU utilization will be 100 percent. In practice, however, it is impossible to achieve this level of CPU utilization due to the cost of context switching between processes and interrupt handling.

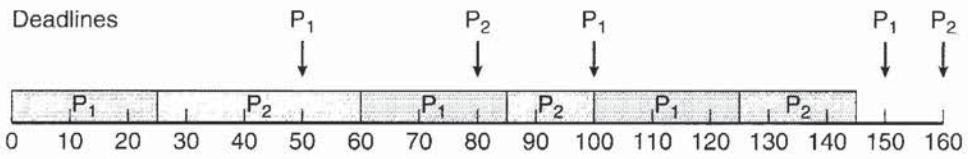


Figure 19.10 Earliest-deadline-first scheduling.

19.5.3 Proportional Share Scheduling

Proportional share schedulers operate by allocating T shares among all applications. An application can receive N shares of time, thus ensuring that the application will have N/T of the total processor time. As an example, assume that there is a total of $T = 100$ shares to be divided among three processes, A, B, and C. A is assigned 50 shares, B is assigned 15 shares, and C is assigned 20 shares. This scheme ensures that A will have 50 percent of total processor time, B will have 15 percent, and C will have 20 percent.

Proportional share schedulers must work in conjunction with an admission control policy to guarantee that an application receives its allocated shares of time. An admission control policy will only admit a client requesting a particular number of shares if there are sufficient shares available. In our current example, we have allocated $50 + 15 + 20 = 75$ shares of the total of 100 shares. If a new process D requested 30 shares, the admission controller would deny D entry into the system.

19.5.4 Pthread Scheduling

The POSIX standard also provides extensions for real-time computing—POSIX.1b. In this section, we cover some of the POSIX Pthread API related to scheduling real-time threads. Pthreads defines two scheduling classes for real-time threads:

- SCHED_FIFO
- SCHED_RR

SCHED_FIFO schedules threads according to a first-come, first-served policy using a FIFO queue as outlined in Section 5.3.1. However, there is no time slicing among threads of equal priority. Therefore, the highest-priority real-time thread at the front of the FIFO queue will be granted the CPU until it terminates or blocks. SCHED_RR (for round-robin) is similar to SCHED_FIFO except that it provides time slicing among threads of equal priority. Pthreads provides an additional scheduling class—SCHED_OTHER—but its implementation is undefined and system specific; it may behave differently on different systems.

The Pthread API specifies the following two functions for getting and setting the scheduling policy:

- `pthread_attr_getsched_policy(pthread_attr_t *attr, int *policy)`
- `pthread_attr_getsched_policy(pthread_attr_t *attr, int policy)`

The first parameter to both functions is a pointer to the set of attributes for the thread. The second parameter is either a pointer to an integer that is set to the current scheduling policy (for `pthread_attr_getsched_policy()`) or an integer value—SCHED_FIFO, SCHED_RR, or SCHED_OTHER—for the `pthread_attr_getsched_policy()` function. Both functions return non-zero values if an error occurs.

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

int main(int argc, char *argv[])
{
   int i, policy;
   pthread_t tid[NUM_THREADS];
   pthread_attr_t attr;

   /* get the default attributes */
   pthread_attr_init(&attr);

   /* get the current scheduling policy */
   if (pthread_attr_getschedpolicy(&attr, &policy) != 0)
      fprintf(stderr, "Unable to get policy.\n");
   else {
      if (policy == SCHED_OTHER)
       printf("SCHED_OTHER\n");
      else if (policy == SCHED_RR)
       printf("SCHED_RR\n");
      else if (policy == SCHED_FIFO)
       printf("SCHED_FIFO\n");
   }

   /* set the scheduling policy - FIFO, RR, or OTHER */
   if (pthread_attr_setschedpolicy(&attr, SCHED_OTHER) != 0)
      fprintf(stderr, "Unable to set policy.\n");

   /* create the threads */
   for (i = 0; i < NUM_THREADS; i++)
       pthread_create(&tid[i],&attr,runner,NULL);

   /* now join on each thread */
   for (i = 0; i < NUM_THREADS; i++)
      pthread_join(tid[i], NULL);
}

/* Each thread will begin control in this function */
void *runner(void *param)
{
   /* do some work ... */

   pthread_exit(0);
}
```

Figure 19.11 Pthread scheduling API.

In Figure 19.11, we illustrate a Pthread program using this API. This program first determines the current scheduling policy followed by setting the scheduling algorithm to SCHED_OTHER.

19.6 VxWorks 5.x

In this section, we describe VxWorks, a popular real-time operating system providing hard real-time support. VxWorks, commercially developed by Wind River Systems, is widely used in automobiles, consumer and industrial devices, and networking equipment such as switches and routers. VxWorks is also used to control the two rovers—*Spirit* and *Opportunity*—that began exploring the planet Mars in 2004.

The organization of VxWorks is shown in Figure 19.12. VxWorks is centered around the *Wind* microkernel. Recall from our discussion in Section 2.7.3 that microkernels are designed so that the operating-system kernel provides a bare minimum of features; additional utilities, such as networking, file systems, and graphics, are provided in libraries outside of the kernel. This approach offers many benefits, including minimizing the size of the kernel—a desirable feature for an embedded system requiring a small footprint.

The Wind microkernel supports the following basic features:

- **Processes**. The Wind microkernel provides support for individual processes and threads (using the Pthread API). However, similar to Linux, VxWorks does not distinguish between processes and threads, instead referring to both as **tasks**.

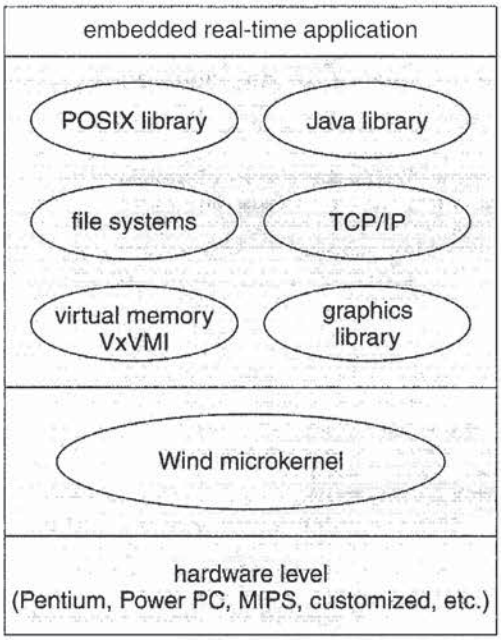


Figure 19.12 The organization of VxWorks.

REAL-TIME LINUX

The Linux operating system is being used increasingly in real-time environments. We have already covered its soft real-time scheduling features (Section 5.6.3), whereby real-time tasks are assigned the highest priority in the system. Additional features in the 2.6 release of the kernel make Linux increasingly suitable for embedded systems. These features include a fully preemptive kernel and a more efficient scheduling algorithm, which runs in $O(1)$ time regardless of the number of tasks active in the system. The 2.6 release also makes it easier to port Linux to different hardware architectures by dividing the kernel into modular components.

Another strategy for integrating Linux into real-time environments involves combining the Linux operating system with a small real-time kernel, thereby providing a system that acts as both a general-purpose and a real-time system. This is the approach taken by the RTLinux operating system. In RTLinux, the standard Linux kernel runs as a task in a small real-time operating system. The real-time kernel handles all interrupts—directing each interrupt to a handler in the standard kernel or to an interrupt handler in the real-time kernel. Furthermore, RTLinux prevents the standard Linux kernel from ever disabling interrupts, thus ensuring that it cannot add latency to the real-time system. RTLinux also provides different scheduling policies, including rate-monotonic scheduling (Section 19.5.1) and earliest-deadline-first scheduling (Section 19.5.2).

- **Scheduling**. Wind provides two separate scheduling models: preemptive and nonpreemptive round-robin scheduling with 256 different priority levels. The scheduler also supports the POSIX API for real-time threads covered in Section 19.5.4.
- **Interrupts**. The Wind microkernel also manages interrupts. To support hard real-time requirements, interrupt and dispatch latency times are bounded.
- **Interprocess communication**. The Wind microkernel provides both shared memory and message passing as mechanisms for communication between separate tasks. Wind also allows tasks to communicate using a technique known as **pipes**—a mechanism that behaves in the same way as a FIFO queue but allows tasks to communicate by writing to a special file, the pipe. To protect data shared by separate tasks, VxWorks provides semaphores and mutex locks with a priority inheritance protocol to prevent priority inversion.

Outside the microkernel, VxWorks includes several component libraries that provide support for POSIX, Java, TCP/IP networking, and the like. All components are optional, allowing the designer of an embedded system to customize the system according to its specific needs. For example, if networking is not required, the TCP/IP library can be excluded from the image of the operating system. Such a strategy allows the operating-system designer to

include only required features, thereby minimizing the size—or footprint—of the operating system.

VxWorks takes an interesting approach to memory management, supporting two levels of virtual memory. The first level, which is quite simple, allows control of the cache on a per-page basis. This policy enables an application to specify certain pages as non-cacheable. When data are being shared by separate tasks running on a multiprocessor architecture, it is possible that shared data can reside in separate caches local to individual processors. Unless an architecture supports a cache-coherency policy to ensure that the same data residing in two caches will not be different, such shared data should not be cached and should instead reside only in main memory so that all tasks maintain a consistent view of the data.

The second level of virtual memory requires the optional virtual memory component VxVMI (Figure 19.12), along with processor support for a memory management unit (MMU). By loading this optional component on systems with an MMU, VxWorks allows a task to mark certain data areas as *private*. A data area marked as private may only be accessed by the task it belongs to. Furthermore, VxWorks allows pages containing kernel code along with the interrupt vector to be declared as read-only. This is useful, as VxWorks does not distinguish between user and kernel modes; all applications run in kernel mode, giving an application access to the entire address space of the system.

19.7 Summary

A real-time system is a computer system requiring that results arrive within a deadline period; results arriving after the deadline has passed are useless. Many real-time systems are embedded in consumer and industrial devices. There are two types of real-time systems: soft and hard real-time systems. Soft real-time systems are the least restrictive, assigning real-time tasks higher scheduling priority than other tasks. Hard real-time systems must guarantee that real-time tasks are serviced within their deadline periods. In addition to strict timing requirements, real-time systems can further be characterized as having only a single purpose and running on small, inexpensive devices.

To meet timing requirements, real-time operating systems must employ various techniques. The scheduler for a real-time operating system must support a priority-based algorithm with preemption. Furthermore, the operating system must allow tasks running in the kernel to be preempted in favor of higher-priority real-time tasks. Real-time operating systems also address specific timing issues by minimizing both interrupt and dispatch latency.

Real-time scheduling algorithms include rate-monotonic and earliest-deadline-first scheduling. Rate-monotonic scheduling assigns tasks that require the CPU more often a higher priority than tasks that require the CPU less often. Earliest-deadline-first scheduling assigns priority according to upcoming deadlines—the earlier the deadline, the higher the priority. Proportional share scheduling uses a technique of dividing up processor time into shares and assigning each process a number of shares, thus guaranteeing each process its proportional share of CPU time. The Pthread API provides various features for scheduling real-time threads as well.

Exercises

19.1 Identify whether hard or soft real-time scheduling is more appropriate in the following environments:

a. Thermostat in a household

b. Control system for a nuclear power plant

c. Fuel economy system in an automobile

d. Landing system in a jet airliner

19.2 Discuss ways in which the priority inversion problem could be addressed in a real-time system. Also discuss whether the solutions could be implemented within the context of a proportional share scheduler.

19.3 The Linux 2.6 kernel can be built with no virtual memory system. Explain how this feature may appeal to designers of real-time systems.

19.4 Under what circumstances is rate-monotonic scheduling inferior to earliest-deadline-first scheduling in meeting the deadlines associated with processes?

19.5 Consider two processes, P_1 and P_2, where $p_1 = 50$, $t_1 = 25$, $p_2 = 75$, and $t_2 = 30$.

a. Can these two processes be scheduled using rate-monotonic scheduling? Illustrate your answer using a Gantt chart.

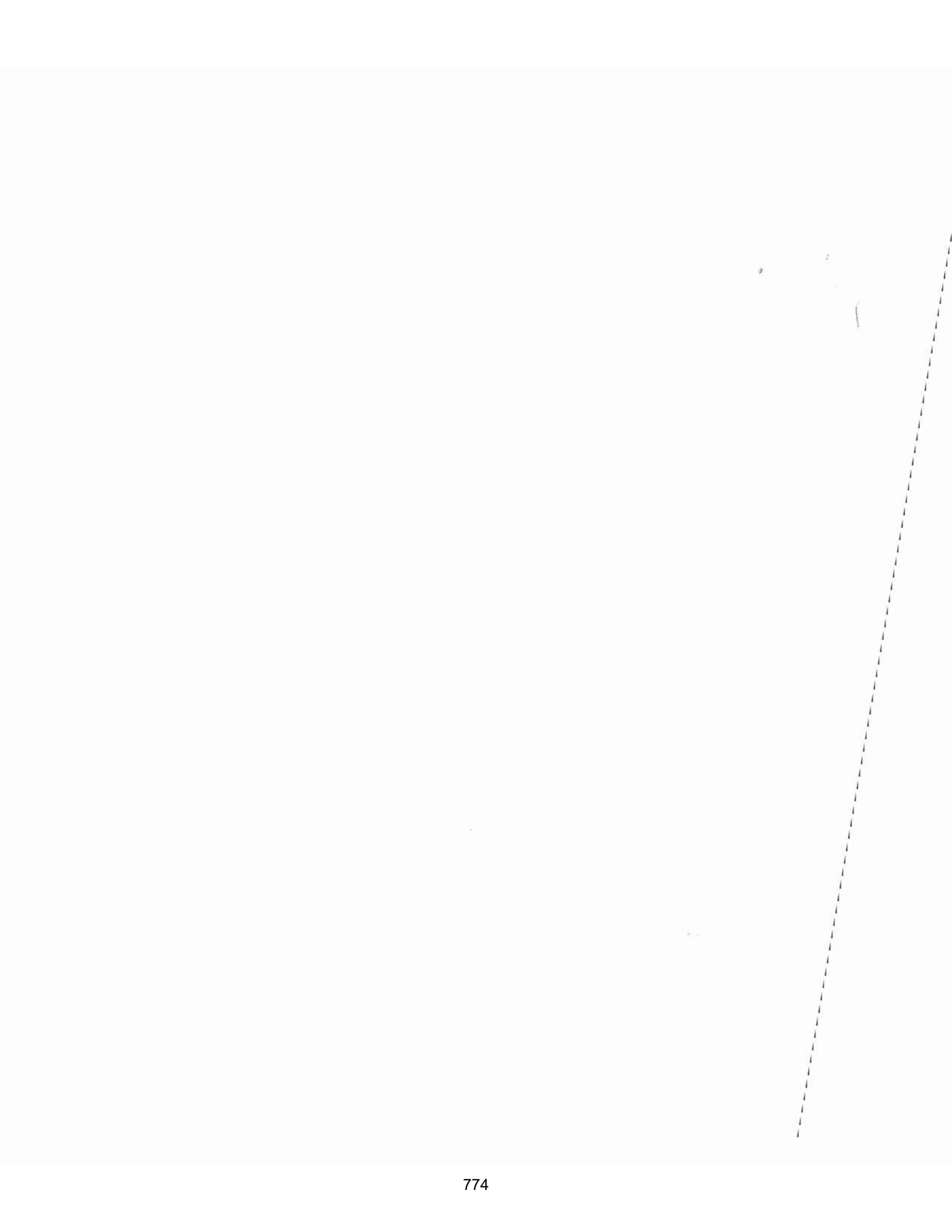
b. Illustrate the scheduling of these two processes using earliest-deadline-first (EDF) scheduling.

19.6 What are the various components of interrupt and dispatch latency?

19.7 Explain why interrupt and dispatch latency times must be bounded in a hard real-time system.

Bibliographical Notes

The scheduling algorithms for hard real-time systems, such as rate monotonic scheduling and earliest-deadline-first scheduling, were presented in Liu and Layland [1973]. Other scheduling algorithms and extensions to previous algorithms were presented in Jensen et al. [1985], Lehoczky et al. [1989], Audsley et al. [1991], Mok [1983], and Stoica et al. [1996]. Mok [1983] described a dynamic priority-assignment algorithm called least-laxity-first scheduling. Stoica et al. [1996] analyzed the proportional share algorithm. Useful information regarding various popular operating systems used in embedded systems can be obtained from http://rtlinux.org, http://windriver.com, and http://qnx.com. Future directions and important research issues in the field of embedded systems were discussed in a research article by Stankovic [1996].



Multimedia Systems



In earlier chapters, we generally concerned ourselves with how operating systems handle conventional data, such as text files, programs, binaries, word-processing documents, and spreadsheets. However, operating systems may have to handle other kinds of data as well. A recent trend in technology is the incorporation of **multimedia data** into computer systems. Multimedia data consist of continuous-media (audio and video) data as well as conventional files. Continuous-media data differ from conventional data in that continuous-media data—such as frames of video—must be delivered (streamed) according to certain time restrictions (for example, 30 frames per second). In this chapter, we explore the demands of continuous-media data. We also discuss in more detail how such data differ from conventional data and how these differences affect the design of operating systems that support the requirements of multimedia systems.

CHAPTER OBJECTIVES

- To identify the characteristics of multimedia data.
- To examine several algorithms used to compress multimedia data.
- To explore the operating-system requirements of multimedia data, including CPU and disk scheduling and network management.

20.1 What Is Multimedia?

The term *multimedia* describes a wide range of applications that are in popular use today. These include audio and video files such as MP3 audio files, DVD movies, and short video clips of movie previews or news stories downloaded over the Internet. Multimedia applications also include live webcasts (broadcast over the World Wide Web) of speeches or sporting events and even live webcams that allow a viewer in Manhattan to observe customers at a cafe in Paris. Multimedia applications need not be either audio or video; rather, a multimedia application often includes a combination of both. For example, a movie may consist of separate audio and video tracks.

Nor must multimedia applications be delivered only to desktop personal computers. Increasingly, they are being directed toward smaller devices, including personal digital assistants (PDAs) and cellular telephones. For example, a stock trader may have stock quotes delivered in real time to her PDA.

In this section, we explore several characteristics of multimedia systems and examine how multimedia files can be delivered from a server to a client system. We also look at common standards for representing multimedia video and audio files.

20.1.1 Media Delivery

Multimedia data are stored in the file system just like any other data. The major difference between a regular file and a multimedia file is that the multimedia file must be accessed at a specific rate, whereas accessing the regular file requires no special timing. Let's use video as an example of what we mean by "rate." Video is represented by a series of images, formally known as **frames**, that are displayed in rapid succession. The faster the frames are displayed, the smoother the video appears. In general, a rate of 24 to 30 frames per second is necessary for video to appear smooth to human eyes. (The eye retains the image of each frame for a short time after it has been presented, a characteristic known as **persistence of vision**. A rate of 24 to 30 frames per second is fast enough to appear continuous.) A rate lower than 24 frames per second will result in a choppy-looking presentation. The video file must be accessed from the file system at a rate consistent with the rate at which the video is being displayed. We refer to data with associated rate requirements as **continuous-media data**.

Multimedia data may be delivered to a client either from the local file system or from a remote server. When the data are delivered from the local file system, we refer to the delivery as **local playback**. Examples include watching a DVD on a laptop computer or listening to an MP3 audio file on a handheld MP3 player. In these cases, the data comprise a regular file that is stored on the local file system and played back (that is, viewed or listened to) from that system.

Multimedia files may also be stored on a remote server and delivered to a client across a network using a technique known as **streaming**. A client may be a personal computer or a smaller device such as a handheld computer, PDA, or cellular telephone. Data from live continuous media—such as live webcams—are also streamed from a server to clients.

There are two types of streaming techniques: progressive download and real-time streaming. With a **progressive download**, a media file containing audio or video is downloaded and stored on the clients local file system. As the file is being downloaded, the client is able to play back the media file without having to wait for the file to be downloaded in its entirety. Because the media file is ultimately stored on the client system, progressive download is most useful for relatively small media files, such as short video clips.

Real-time streaming differs from progressive download in that the media file is streamed to the client but is only played—and not stored—by the client. Because the media file is not stored on the client system, real-time streaming is preferable to progressive download for media files that might be too large

for storage on the system, such as long videos and Internet radio and TV broadcasts.

Both progressive download and real-time streaming may allow a client to move to different points in the stream, just as you can use the fast-forward and rewind operations on a VCR controller to move to different points in the VCR tape. For example, we could move to the end of a 5-minute streaming video or replay a certain section of a movie clip. The ability to move around within the media stream is known as **random access**.

Two types of real-time streaming are available: live streaming and on-demand streaming. **Live streaming** is used to deliver an event, such as a concert or a lecture, live as it is actually occurring. A radio program broadcast over the Internet is an example of a live real-time stream. In fact, one of the authors of this text regularly listens to a favorite radio station from Vermont while at his home in Utah as it is streamed live over the Internet. Live real-time streaming is also used for applications such as live webcams and video conferencing. Due to its live delivery, this type of real-time streaming does not allow clients random access to different points in the media stream. In addition, live delivery means that a client who wishes to view (or listen to) a particular live stream already in progress will "join" the session "late," thereby missing earlier portions of the stream. The same thing happens with a live TV or radio broadcast. If you start watching the 7:00 P.M. news at 7:10 P.M., you will have missed the first 10 minutes of the broadcast.

On-demand streaming is used to deliver media streams such as full-length movies and archived lectures. The difference between live and on-demand streaming is that on-demand streaming does not take place as the event is occurring. Thus, for example, whereas watching a live stream is like watching a news broadcast on TV, watching an on-demand stream is like viewing a movie on a DVD player at some convenient time—there is no notion of arriving late. Depending on the type of on-demand streaming, a client may or may not have random access to the stream.

Examples of well-known streaming media products include RealPlayer, Apple QuickTime, and Windows Media Player. These products include both servers that stream the media and client media players that are used for playback.

20.1.2 Characteristics of Multimedia Systems

The demands of multimedia systems are unlike the demands of traditional applications. In general, multimedia systems may have the following characteristics:

1. Multimedia files can be quite large. For example, a 100-minute MPEG-1 video file requires approximately 1.125 GB of storage space; 100 minutes of high-definition television (HDTV) requires approximately 15 GB of storage. A server storing hundreds or thousands of digital video files may thus require several terabytes of storage.
2. Continuous media may require very high data rates. Consider digital video, in which a frame of color video is displayed at a resolution of 800×600. If we use 24 bits to represent the color of each pixel (which allows us to have 2^{24}, or roughly 16 million, different colors), a single

frame requires $800 \times 600 \times 24 = 11,520,000$ bits of data. If the frames are displayed at a rate of 30 frames per second, a bandwidth in excess of 345 Mbps is required.

3. Multimedia applications are sensitive to timing delays during playback. Once a continuous-media file is delivered to a client, delivery must continue at a certain rate during playback of the media; otherwise, the listener or viewer will be subjected to pauses during the presentation.

20.1.3 Operating-System Issues

For a computer system to deliver continuous-media data, it must guarantee the specific rate and timing requirements—also known as **quality of service**, or QoS, requirements—of continuous media.

Providing these QoS guarantees affects several components in a computer system and influences such operating-system issues as CPU scheduling, disk scheduling, and network management. Specific examples include the following:

1. Compression and decoding may require significant CPU processing.
2. Multimedia tasks must be scheduled with certain priorities to ensure meeting the deadline requirements of continuous media.
3. Similarly, file systems must be efficient to meet the rate requirements of continuous media.
4. Network protocols must support bandwidth requirements while minimizing delay and jitter.

In later sections, we explore these and several other issues related to QoS. First, however, we provide an overview of various techniques for compressing multimedia data. As suggested above, compression makes significant demands on the CPU.

20.2 Compression

Because of the size and rate requirements of multimedia systems, multimedia files are often compressed from their original form to a much smaller form. Once a file has been compressed, it takes up less space for storage and can be delivered to a client more quickly. Compression is particularly important when the content is being streamed across a network connection. In discussing file compression, we often refer to the **compression ratio**, which is the ratio of the original file size to the size of the compressed file. For example, an 800-KB file that is compressed to 100 KB has a compression ratio of 8:1.

Once a file has been compressed (**encoded**), it must be decompressed (**decoded**) before it can be accessed. A feature of the algorithm used to compress the file affects the later decompression. Compression algorithms are classified as either **lossy** or **lossless**. With lossy compression, some of the original data are lost when the file is decoded, whereas lossless compression ensures that the compressed file can always be restored back to its original form. In general, lossy techniques provide much higher compression ratios. Obviously, though,

only certain types of data can tolerate lossy compression—namely, images, audio, and video. Lossy compression algorithms often work by eliminating certain data, such as very high or low frequencies that a human ear cannot detect. Some lossy compression algorithms used on video operate by storing only the differences between successive frames. Lossless algorithms are used for compressing text files, such as computer programs (for example, **zipping** files), because we want to restore these compressed files to their original state.

A number of different lossy compression schemes for continuous-media data are commercially available. In this section, we cover one used by the Moving Picture Experts Group, better known as MPEG.

MPEG refers to a set of file formats and compression standards for digital video. Because digital video often contains an audio portion as well, each of the standards is divided into three layers. Layers 3 and 2 apply to the audio and video portions of the media file. Layer 1 is known as the **systems layer** and contains timing information to allow the MPEG player to multiplex the audio and video portions so that they are synchronized during playback. There are three major MPEG standards: MPEG-1, MPEG-2, and MPEG-4.

MPEG-1 is used for digital video and its associated audio stream. The resolution of MPEG-1 is 352×240 at 30 frames per second with a bit rate of up to 1.5 Mbps. This provides a quality slightly lower than that of conventional VCR videos. MP3 audio files (a popular medium for storing music) use the audio layer (layer 3) of MPEG-1. For video, MPEG-1 can achieve a compression ratio of up to 200:1, although in practice compression ratios are much lower. Because MPEG-1 does not require high data rates, it is often used to download short video clips over the Internet.

MPEG-2 provides better quality than MPEG-1 and is used for compressing DVD movies and digital television (including high-definition television, or HDTV). MPEG-2 identifies a number of **levels** and **profiles** of video compression. The level refers to the resolution of the video; the profile characterizes the video's quality. In general, the higher the level of resolution and the better the quality of the video, the higher the required data rate. Typical bit rates for MPEG-2 encoded files are 1.5 Mbps to 15 Mbps. Because MPEG-2 requires higher rates, it is often unsuitable for delivery of video across a network and is generally used for local playback.

MPEG-4 is the most recent of the standards and is used to transmit audio, video, and graphics, including two-dimensional and three-dimensional animation layers. Animation makes it possible for end users to interact with the file during playback. For example, a potential home buyer can download an MPEG-4 file and take a virtual tour through a home she is considering purchasing, moving from room to room as she chooses. Another appealing feature of MPEG-4 is that it provides a scalable level of quality, allowing delivery over relatively slow network connections such as 56-Kbps modems or over high-speed local area networks with rates of several megabits per second. Furthermore, by providing a scalable level of quality, MPEG-4 audio and video files can be delivered to wireless devices, including handheld computers, PDAs, and cell phones.

All three MPEG standards discussed here perform lossy compression to achieve high compression ratios. The fundamental idea behind MPEG compression is to store the differences between successive frames. We do not cover further details of how MPEG performs compression but rather encourage

the interested reader to consult the bibliographical notes at the end of this chapter.

20.3 Requirements of Multimedia Kernels

As a result of the characteristics described in Section 20.1.2, multimedia applications often require levels of service from the operating system that differ from the requirements of traditional applications, such as word processors, compilers, and spreadsheets. Timing and rate requirements are perhaps the issues of foremost concern, as the playback of audio and video data demands that the data be delivered within a certain deadline and at a continuous, fixed rate. Traditional applications typically do not have such time and rate constraints.

Tasks that request data at constant intervals—or **periods**—are known as **periodic processes**. For example, an MPEG-1 video might require a rate of 30 frames per second during playback. Maintaining this rate requires that a frame be delivered approximately every $1/30^{th}$ or 3.34 hundredths of a second. To put this in the context of deadlines, let's assume that frame F_j succeeds frame F_i in the video playback and that frame F_i was displayed at time T_0. The deadline for displaying frame F_j is 3.34 hundredths of a second after time T_0. If the operating system is unable to display the frame by this deadline, the frame will be omitted from the stream.

As mentioned earlier, rate requirements and deadlines are known as quality of service (QoS) requirements. There are three QoS levels:

1. **Best-effort service**. The system makes a best-effort attempt to satisfy the requirements; however, no guarantees are made.
2. **Soft QoS**. This level treats different types of traffic in different ways, giving certain traffic streams higher priority than other streams. However, just as with best-effort service, no guarantees are made.
3. **Hard QoS**. The quality-of-service requirements are guaranteed.

Traditional operating systems—the systems we have discussed in this text so far—typically provide only best-effort service and rely on **overprovisioning**; that is, they simply assume that the total amount of resources available will tend to be larger than a worst-case workload would demand. If demand exceeds resource capacity, manual intervention must take place, and a process (or several processes) must be removed from the system. However next-generation multimedia systems cannot make such assumptions. These systems must provide continuous-media applications with the guarantees made possible by hard QoS. Therefore, in the remainder of this discussion, when we refer to QoS, we mean hard QoS. Next, we explore various techniques that enable multimedia systems to provide such service-level guarantees.

There are a number of parameters defining QoS for multimedia applications, including the following:

- **Throughput**. Throughput is the total amount of work done during a certain interval. For multimedia applications, throughput is the required data rate.

- **Delay**. Delay refers to the elapsed time from when a request is first submitted to when the desired result is produced. For example, the time from when a client requests a media stream to when the stream is delivered is the delay.
- **Jitter**. Jitter is related to delay; but whereas delay refers to the time a client must wait to receive a stream, jitter refers to delays that occur during playback of the stream. Certain multimedia applications, such as on-demand real-time streaming, can tolerate this sort of delay. Jitter is generally considered unacceptable for continuous-media applications, however, because it may mean long pauses—or lost frames—during playback. Clients can often compensate for jitter by buffering a certain amount of data—say, 5 seconds' worth—before beginning playback.
- **Reliability**. Reliability refers to how errors are handled during transmission and processing of continuous media. Errors may occur due to lost packets in the network or processing delays by the CPU. In these—and other—scenarios, errors cannot be corrected, since packets typically arrive too late to be useful.

The quality of service may be **negotiated** between the client and the server. For example, continuous-media data may be compressed at different levels of quality: the higher the quality, the higher the required data rate. A client may negotiate a specific data rate with a server, thus agreeing to a certain level of quality during playback. Furthermore, many media players allow the client to configure the player according to the speed of the client's connection to the network. This allows a client to receive a streaming service at a data rate specific to a particular connection. Thus, the client is negotiating quality of service with the content provider.

To provide QoS guarantees, operating systems often use **admission control**, which is simply the practice of admitting a request for service only if the server has sufficient resources to satisfy the request. We see admission control quite often in our everyday lives. For example, a movie theater only admits as many customers as it has seats in the theater. (There are also many situations in everyday life where admission control is not practiced but would be desirable!) If no admission control policy is used in a multimedia environment, the demands on the system might become so great that the system becomes unable to meet its QoS guarantees.

In Chapter 6, we discussed using semaphores as a method of implementing a simple admission control policy. In this scenario, there exist a finite number of non-shareable resources. When a resource is requested, we will only grant the request if there are sufficient resources available; otherwise the requesting process is forced to wait until a resource becomes available. Semaphores may be used to implement an admission control policy by first initializing a semaphore to the number of resources available. Every request for a resource is made through a `wait()` operation on the semaphore; a resource is released with an invocation of `signal()` on the semaphore. Once all resources are in use, subsequent calls to `wait()` block until there is a corresponding `signal()`.

A common technique for implementing admission control is to use **resource reservations**. For example, resources on a file server may include the CPU, memory, file system, devices, and network (Figure 20.1). Note that

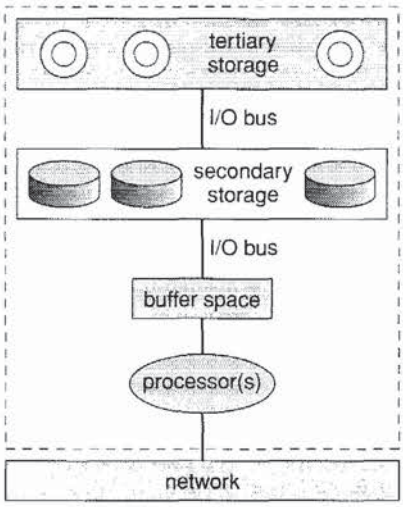


Figure 20.1 Resources on a file server.

resources may be either exclusive or shared and that there may be either single or multiple instances of each resource type. To use a resource, a client must make a reservation request for the resource in advance. If the request cannot be granted, the reservation is denied. An admission control scheme assigns a **resource manager** to each type of resource. Requests for resources have associated QoS requirements—for example, required data rates. When a request for a resource arrives, the resource manager determines if the resource can meet the QoS demands of the request. If not, the request may be rejected, or a lower level of QoS may be negotiated between the client and the server. If the request is accepted, the resource manager reserves the resources for the requesting client, thus assuring the client the desired QoS requirements. In Section 20.7.2, we examine the admission control algorithm used to ensure QoS guarantees in the CineBlitz multimedia storage server.

20.4 CPU Scheduling

In Chapter 19, which covered real-time systems, we distinguished between **soft real-time systems** and **hard real-time systems**. Soft real-time systems simply give scheduling priority to critical processes. A soft real-time system ensures that a critical process will be given preference over a noncritical process but provides no guarantee as to when the critical process will be scheduled. A typical requirement of continuous media, however, is that data must be delivered to a client by a certain deadline; data that do not arrive by the deadline are unusable. Multimedia systems thus require hard real-time scheduling to ensure that a critical task will be serviced within a guaranteed period of time.

Another scheduling issue concerns whether a scheduling algorithm uses **static priority** or **dynamic priority**—a distinction we first discussed in Chapter 5. The difference between the two is that the priority of a process will remain unchanged if the scheduler assigns it a static priority. Scheduling algorithms

that assign dynamic priorities allow priorities to change over time. Most operating systems use dynamic priorities when scheduling non-real-time tasks with the intention of giving higher priority to interactive processes. However, when scheduling real-time tasks, most systems assign static priorities, as the design of the scheduler is less complex.

Several of the real-time scheduling strategies discussed in Section 19.5 can be used to meet the rate and deadline QoS requirements of continuous-media applications.

20.5 Disk Scheduling

We first discussed disk scheduling in Chapter 12. There, we focused primarily on systems that handle conventional data; for these systems, the scheduling goals are fairness and throughput. As a result, most traditional disk schedulers employ some form of the SCAN (Section 12.4.3) or C-SCAN (Section 12.4.4) algorithm.

Continuous-media files, however, have two constraints that conventional data files generally do not have: timing deadlines and rate requirements. These two constraints must be satisfied to preserve QoS guarantees, and disk-scheduling algorithms must be optimized for the constraints. Unfortunately, these two constraints are often in conflict. Continuous-media files typically require very high disk-bandwidth rates to satisfy their data-rate requirements. Because disks have relatively low transfer rates and relatively high latency rates, disk schedulers must reduce the latency times to ensure high bandwidth. However, reducing latency times may result in a scheduling policy that does not prioritize according to deadlines. In this section, we explore two disk-scheduling algorithms that meet the QoS requirements for continuous-media systems.

20.5.1 Earliest-Deadline-First Scheduling

We first saw the earliest-deadline-first (EDF) algorithm in Section 19.5.2 as an example of a CPU-scheduling algorithm that assigns priorities according to deadlines. EDF can also be used as a disk-scheduling algorithm; in this context, EDF uses a queue to order requests according to the time each request must be completed (its deadline). EDF is similar to shortest-seek-time-first (SSTF), which was discussed in 12.4.2, except that instead of servicing the request closest to the current cylinder, we service requests according to deadline—the request with the closest deadline is serviced first.

A problem with this approach is that servicing requests strictly according to deadline may result in higher seek times, since the disk heads may move randomly throughout the disk without any regard to their current position. For example, suppose a disk head is currently at cylinder 75 and the queue of cylinders (ordered according to deadlines) is 98, 183, 105. Under strict EDF scheduling, the disk head will move from 75, to 98, to 183, and then back to 105. Note that the head passes over cylinder 105 as it travels from 98 to 183. It is possible that the disk scheduler could have serviced the request for cylinder 105 en route to cylinder 183 and still preserved the deadline requirement for cylinder 183.

20.5.2 SCAN-EDF Scheduling

The fundamental problem with strict EDF scheduling is that it ignores the position of the read-write heads of the disk; it is possible that the movement of the heads will swing wildly to and fro across the disk, leading to unacceptable seek times that negatively affect disk throughput. Recall that this is the same issue faced with FCFS scheduling (Section 12.4.1). We ultimately addressed this issue by adopting SCAN scheduling, wherein the disk arm moves in one direction across the disk, servicing requests according to their proximity to the current cylinder. Once the disk arm reaches the end of the disk, it begins moving in the reverse direction. This strategy optimizes seek times.

SCAN-EDF is a hybrid algorithm that combines EDF with SCAN scheduling. SCAN-EDF starts with EDF ordering but services requests with the same deadline using SCAN order. What if several requests have different deadlines that are relatively close together? In this case, SCAN-EDF may batch requests, using SCAN ordering to service requests in the same batch. There are many techniques for batching requests with similar deadlines; the only requirement is that reordering requests within a batch must not prevent a request from being serviced by its deadline. If deadlines are equally distributed, batches can be organized in groups of a certain size—say, 10 requests per batch.

Another approach is to batch requests whose deadlines fall within a given time threshold—say, 100 milliseconds. Let's consider an example in which we batch requests in this way. Assume we have the following requests, each with a specified deadline (in milliseconds) and the cylinder being requested:

request	deadline	cylinder
A	150	25
B	201	112
C	399	95
D	94	31
E	295	185
F	78	85
G	165	150
H	125	101
I	300	85
J	210	90

Suppose we are at $time_0$, the cylinder currently being serviced is 50, and the disk head is moving toward cylinder 51. According to our batching scheme, requests D and F will be in the first batch; A, G, and H in batch 2; B, E, and J in batch 3; and C and I in the last batch. Requests within each batch will be ordered according to SCAN order. Thus, in batch 1, we will first service request F and then request D. Note that we are moving downward in cylinder numbers, from 85 to 31. In batch 2, we first service request A; then the heads begin moving upward in cylinders, servicing requests H and then G. Batch 3 is serviced in the order E, B, J. Requests I and C are serviced in the final batch.

20.6 Network Management

Perhaps the foremost QoS issue with multimedia systems concerns preserving rate requirements. For example, if a client wishes to view a video compressed with MPEG-1, the quality of service greatly depends on the system's ability to deliver the frames at the required rate..

Our coverage of issues such as CPU- and disk-scheduling algorithms has focused on how these techniques can be used to better meet the quality-of-service requirements of multimedia applications. However, if the media file is being streamed over a network—perhaps the Internet—issues relating to how the network delivers the multimedia data can also significantly affect how QoS demands are met. In this section, we explore several network issues related to the unique demands of continuous media.

Before we proceed, it is worth noting that computer networks in general —and the Internet in particular— currently do not provide network protocols that can ensure the delivery of data with timing requirements. (There are some proprietary protocols—notably those running on Cisco routers—that do allow certain network traffic to be prioritized to meet QoS requirements. Such proprietary protocols are not generalized for use across the Internet and therefore do not apply to our discussion.)

When data are routed across a network, it is likely that the transmission will encounter congestion, delays, and other network traffic issues—issues that are beyond the control of the originator of the data. For multimedia data with timing requirements, any timing issues must be synchronized between the end hosts: the server delivering the content and the client playing it back.

One protocol that addresses timing issues is the **real-time transport protocol** (RTP). RTP is an Internet standard for delivering real-time data, including audio and video. It can be used for transporting media formats such as MP3 audio files and video files compressed using MPEG. RTP does not provide any QoS guarantees; rather, it provides features that allow a receiver to remove jitter introduced by delays and congestion in the network.

In following sections, we consider two other approaches for handling the unique requirements of continuous media.

20.6.1 Unicasting and Multicasting

In general, there are three methods for delivering content from a server to a client across a network:

- **Unicasting**. The server delivers the content to a single client. If the content is being delivered to more than one client, the server must establish a separate unicast for each client.
- **Broadcasting**. The server delivers the content to all clients, regardless of whether they wish to receive the content or not.
- **Multicasting**. The server delivers the content to a group of receivers who indicate they wish to receive the content; this method lies somewhere between unicasting and broadcasting.

An issue with unicast delivery is that the server must establish a separate unicast session for each client. This seems especially wasteful for live real-time

streaming, where the server must make several copies of the same content, one for each client. Obviously, broadcasting is not always appropriate, as not all clients may wish to receive the stream. (Suffice to say that broadcasting is typically only used across local area networks and is not possible across the public Internet.)

Multicasting appears to be a reasonable compromise, since it allows the server to deliver a single copy of the content to all clients indicating that they wish to receive it. The difficulty with multicasting from a practical standpoint is that the clients must be physically close to the server or to intermediate routers that relay the content from the originating server. If the route from the server to the client must cross intermediate routers, the routers must also support multicasting. If these conditions are not met, the delays incurred during routing may result in violation of the timing requirements of the continuous media. In the worst case, if a client is connected to an intermediate router that does not support multicasting, the client will be unable to receive the multicast stream at all!

Currently, most streaming media are delivered across unicast channels; however, multicasting is used in various areas where the organization of the server and clients is known in advance. For example, a corporation with several sites across a country may be able to ensure that all sites are connected to multicasting routers and are within reasonable physical proximity to the routers. The organization will then be able to deliver a presentation from the chief executive officer using multicasting.

20.6.2 Real-Time Streaming Protocol

In Section 20.1.1, we described some features of streaming media. As we noted there, users may be able to randomly access a media stream, perhaps rewinding or pausing, as they would with a VCR controller. How is this possible?

To answer this question, let's consider how streaming media are delivered to clients. One approach is to stream the media from a standard web server using the hypertext transport protocol, or HTTP—the protocol used to deliver

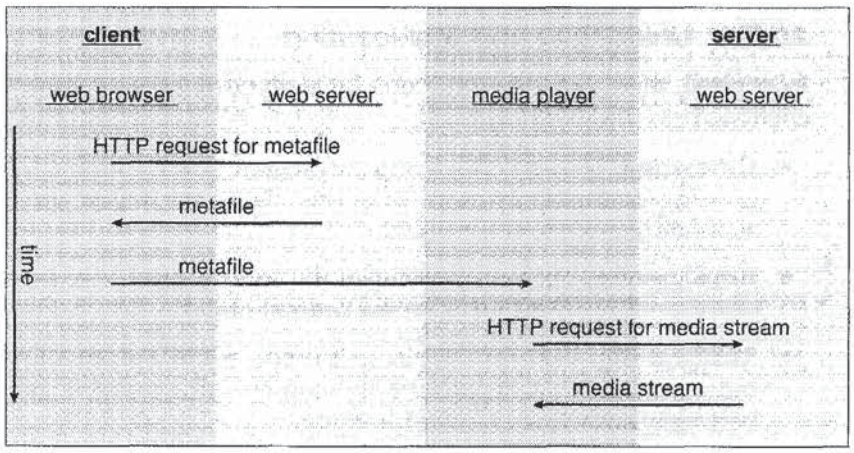


Figure 20.2 Streaming media from a conventional web server.

documents from a web server. Quite often, clients use a **media player**, such as QuickTime, RealPlayer, or Windows Media Player, to play back media streamed from a standard web server. Typically, the client first requests a **metafile**, which contains the location (possibly identified by a uniform resource locator, or URL) of the streaming media file. This metafile is delivered to the client's web browser, and the browser then starts the appropriate media player according to the type of media specified by the metafile. For example, a Real Audio stream would require the RealPlayer, while the Windows Media Player would be used to play back streaming Windows media. The media player then contacts the web server and requests the streaming media. The stream is delivered from the web server to the media player using standard HTTP requests. This process is outlined in Figure 20.2.

The problem with delivering streaming media from a standard web server is that HTTP is considered a **stateless** protocol; thus, a web server does not maintain the state (or status) of its connection with a client. As a result, it is difficult for a client to pause during the delivery of streaming media content, since pausing would require the web server to know where in the stream to begin when the client wished to resume playback.

An alternative strategy is to use a specialized streaming server that is designed specifically for streaming media. One protocol designed for communication between streaming servers and media players is known as the real-time streaming protocol, or RTSP. The significant advantage RTSP provides over HTTP is a stateful connection between the client and the server, which allows the client to pause or seek to random positions in the stream during playback. Delivery of streaming media using RTSP is similar to delivery using HTTP (Figure 20.2) in that the meta file is delivered using a conventional web server. However, rather than using a web server, the streaming media is delivered from a streaming server using the RTSP protocol. The operation of RTSP is shown in Figure 20.3.

RTSP defines several commands as part of its protocol; these commands are sent from a client to an RTSP streaming server. The commands include:

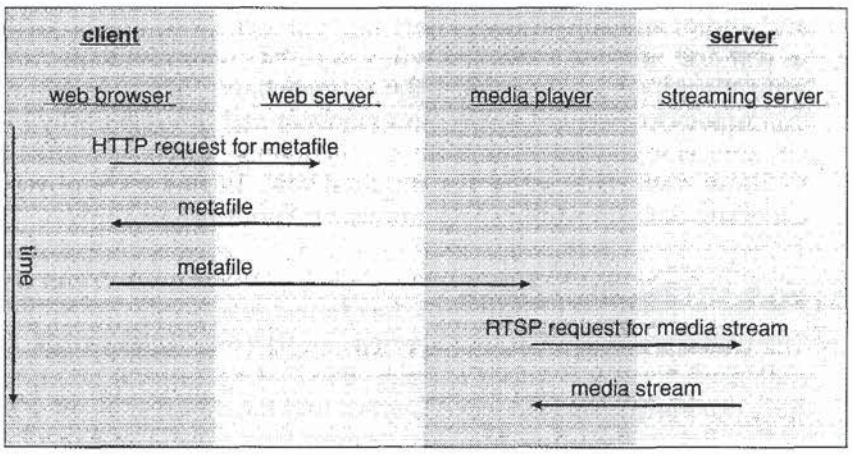


Figure 20.3 Real-time streaming protocol (RTSP).

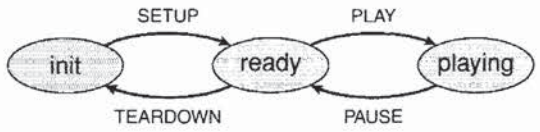


Figure 20.4 Finite-state machine representing RTSP.

- **SETUP**. The server allocates resources for a client session.
- **PLAY**. The server delivers a stream to a client session established from a SETUP command.
- **PAUSE**. The server suspends delivery of a stream but maintains the resources for the session.
- **TEARDOWN**. The server breaks down the connection and frees up resources allocated for the session.

The commands can be illustrated with a state machine for the server, as shown in Figure 20.4. As you can see in the figure, the RTSP server may be in one of three states: **init**, **ready**, and **playing**. Transitions between these three states are triggered when the server receives one of the RTSP commands from the client.

Using RTSP rather than HTTP for streaming media offers several other advantages, but they are primarily related to networking issues and are therefore beyond the scope of this text. We encourage interested readers to consult the bibliographical notes at the end of this chapter for sources of further information.

20.7 An Example: CineBlitz

The CineBlitz multimedia storage server is a high-performance media server that supports both continuous media with rate requirements (such as video and audio) and conventional data with no associated rate requirements (such as text and images). CineBlitz refers to clients with rate requirements as **real-time clients**, whereas **non-real-time clients** have no rate constraints. CineBlitz guarantees to meet the rate requirements of real-time clients by implementing an admission controller, admitting a client only if there are sufficient resources to allow data retrieval at the required rate. In this section, we explore the CineBlitz disk-scheduling and admission-control algorithms.

20.7.1 Disk Scheduling

The CineBlitz disk scheduler services requests in **cycles**. At the beginning of each service cycle, requests are placed in C-SCAN order (Section 12.4.4). Recall from our earlier discussions of C-SCAN that the disk heads move from one end of the disk to the other. However, rather than reversing direction when they reach the end of the disk, as in pure SCAN disk scheduling (Section 12.4.3), the disk heads move back to the beginning of the disk.

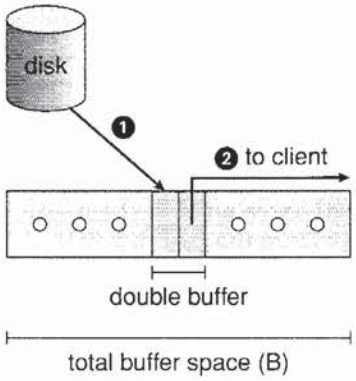


Figure 20.5 Double buffering in CineBlitz.

20.7.2 Admission Control

The admission-control algorithm in CineBlitz must monitor requests from both real-time and non-real-time clients, ensuring that both classes of clients receive service. Furthermore, the admission controller must provide the rate guarantees required by real-time clients. To ensure fairness, only a fraction p of time is reserved for real-time clients, while the remainder, $1 - p$, is set aside for non-real-time clients. Here, we explore the admission controller for real-time clients only; thus, the term *client* refers to a real-time client.

The admission controller in CineBlitz monitors various system resources, such as disk bandwidth and disk latency, while keeping track of available buffer space. The CineBlitz admission controller admits a client only if there is enough available disk bandwidth and buffer space to retrieve data for the client at its required rate.

CineBlitz queues requests R_1, R_2, R_3, ... R_n for continuous media files where r_i is the required data rate for a given request R_i. Requests in the queue are served in cyclic order using a technique known as **double buffering**, wherein a buffer is allocated for each request R_i of size $2 \times T \times r_i$.

During each cycle I, the server must:

1. Retrieve the data from disk to buffer (I mod 2).
2. Transfer data from the (($I + 1$) mod 2) buffer to the client.

This process is illustrated in Figure 20.5. For N clients, the total buffer space B required is

$$\sum_{i=1}^{N} 2 \times T \times r_i \leq B. \tag{20.1}$$

The fundamental idea behind the admission controller in CineBlitz is to bound requests for entry into the queue according to the following criteria:

1. The service time for each request is first estimated.
2. A request is admitted only if the sum of the estimated service times for all admitted requests does not exceed the duration of service cycle T.

Let $T \times r_i$ bits be retrieved during a cycle for each real-time client R_i with rate r_i. If $R_1, R_2, \ldots R_n$ are the clients currently active in the system, then the admission controller must ensure that the total times for retrieving $T \times r_1, T \times r_2, \ldots, T \times r_n$ bits for the corresponding real-time clients does not exceed T. We explore the details of this admission policy in the remainder of this section.

If b is the size of a disk block, then the maximum number of disk blocks that can be retrieved for request R_k during each cycle is $\lceil (T \times r_k)/b \rceil + 1$. The 1 in this formula comes from the fact that, if $T \times r_k$ is less than b, then it is possible for $T \times r_k$ bits to span the last portion of one disk block and the beginning of another, causing two blocks to be retrieved. We know that the retrieval of a disk block involves (a) a seek to the track containing the block and (b) the rotational delay as the data in the desired track arrives under the disk head. As described, CineBlitz uses a C-SCAN disk-scheduling algorithm, so disk blocks are retrieved in the sorted order of their positions on the disk.

If t_{seek} and t_{rot} refer to the worst-case seek and rotational delay times, the maximum latency incurred for servicing N requests is

$$2 \times t_{seek} + \sum_{i=1}^{N} \left(\lceil \frac{T \times r_i}{b} \rceil + 1 \right) \times t_{rot}. \tag{20.2}$$

In this equation, the $2 \times t_{seek}$ component refers to the maximum disk-seek latency incurred in a cycle. The second component reflects the sum of the retrievals of the disk blocks multiplied by the worst-case rotational delay.

If the transfer rate of the disk is r_{disk}, then the time to transfer $T \times r_k$ bits of data for request R_k is $(T \times r_k)/r_{disk}$. As a result, the total time for retrieving $T \times r_1, T \times r_2, \ldots, T \times r_n$ bits for requests $R_1, R_2, \ldots, R_n$ is the sum of equation 20.2 and

$$\sum_{i=1}^{N} \frac{T \times r_i}{r_{disk}} \tag{20.3}$$

Therefore, the admission controller in CineBlitz only admits a new client R_i if at least $2 \times T \times r_i$ bits of free buffer space are available for the client and the following equation is satisfied:

$$2 \times t_{seek} + \sum_{i=1}^{N} \left(\lceil \frac{T \times r_i}{b} \rceil + 1 \right) \times t_{rot} + \sum_{i=1}^{N} \frac{T \times r_i}{r_{disk}} \leq T. \tag{20.4}$$

20.8 Summary

Multimedia applications are in common use in modern computer systems. Multimedia files include video and audio files, which may be delivered to systems such as desktop computers, personal digital assistants, and cell phones. The primary distinction between multimedia data and conventional data is that multimedia data have specific rate and deadline requirements. Because multimedia files have specific timing requirements, the data must often be compressed before delivery to a client for playback. Multimedia data may be delivered either from the local file system or from a multimedia server across a network connection using a technique known as streaming.

The timing requirements of multimedia data are known as quality-of-service requirements, and conventional operating systems often cannot make quality-of-service guarantees. To provide quality of service, multimedia systems must provide a form of admission control whereby a system accepts a request only if it can meet the quality-of-service level specified by the request. Providing quality-of-service guarantees requires evaluating how an operating system performs CPU scheduling, disk scheduling, and network management. Both CPU and disk scheduling typically use the deadline requirements of a continuous-media task as a scheduling criterion. Network management requires the use of protocols that handle delay and jitter caused by the network as well as allowing a client to pause or move to different positions in the stream during playback.

Exercises

20.1 Provide examples of multimedia applications that are delivered over the Internet.

20.2 Distinguish between progressive download and real-time streaming.

20.3 Which of the following types of real-time streaming applications can tolerate delay? Which can tolerate jitter?

- Live real-time streaming
- On-demand real-time streaming

20.4 Discuss what techniques could be used to meet quality-of-service requirements for multimedia applications in the following components of a system:

- Process scheduler
- Disk scheduler
- Memory manager

20.5 Explain why the traditional Internet protocols for transmitting data are not sufficient to provide the quality-of-service guarantees required for a multimedia system. Discuss what changes are required to provide the QoS guarantees.

20.6 Assume that a digital video file is being displayed at a rate of 30 frames per second; the resolution of each frame is 640×480, and 24 bits are being used to represent each color. Assuming that no compression is being used, what is the bandwidth necessary to deliver this file? Next, assuming that the file has been compressed at a ratio of 200 : 1, what is the bandwidth necessary to deliver the compressed file?

20.7 A multimedia application consists of a set containing 100 images, 10 minutes of video, and 10 minutes of audio. The compressed sizes of the images, video, and audio are 500 MB, 550 MB, and 8 MB, respectively. The images were compressed at a ratio of 15 : 1, and the video and

audio were compressed at 200 : 1 and 10 : 1, respectively. What were the sizes of the images, video, and audio before compression?

20.8 Assume that we wish to compress a digital video file using MPEG-1 technology. The target bit rate is 1.5 Mbps. If the video is displayed at a resolution of 352 × 240 at 30 frames per second using 24 bits to represent each color, what is the necessary compression ratio to achieve the desired bit rate?

20.9 Consider two processes, P_1 and P_2, where $p_1 = 50$, $t_1 = 25$, $p_2 = 75$, and $t_2 = 30$.

a. Can these two processes be scheduled using rate-monotonic scheduling? Illustrate your answer using a Gantt chart.

b. Illustrate the scheduling of these two processes using earliest-deadline-first (EDF) scheduling.

20.10 The following table contains a number of requests with their associated deadlines and cylinders. Requests with deadlines occurring within 100 milliseconds of each other will be batched. The disk head is currently at cylinder 94 and is moving toward cylinder 95. If SCAN-EDF disk scheduling is used, how are the requests batched together, and what is the order of requests within each batch?

request	deadline	cylinder
R1	57	77
R2	300	95
R3	250	25
R4	88	28
R5	85	100
R6	110	90
R7	299	50
R8	300	77
R9	120	12
R10	212	2

20.11 Repeat the preceding question, but this time batch requests that have deadlines occurring within 75 milliseconds of each other.

20.12 Contrast unicasting, multicasting, and broadcasting as techniques for delivering content across a computer network.

20.13 Describe why HTTP is often insufficient for delivering streaming media.

20.14 What operating principle is used by the CineBlitz system in performing admission control for requests for media files?

Bibliographical Notes

Fuhrt [1994] provides a general overview of multimedia systems. Topics related to the delivery of multimedia through networks can be found in Kurose and Ross [2005]. Operating-system support for multimedia is discussed in Steinmetz [1995] and Leslie et al. [1996]. Resource management for resources such as processing capability and memory buffers are discussed in Mercer et al. [1994] and Druschel and Peterson [1993]. Reddy and Wyllie [1994] give a good overview of issues relating to the use of I/O in a multimedia system. Discussions regarding the appropriate programming model for developing multimedia applications are presented in Regehr et al. [2000]. An admission control system for a rate-monotonic scheduler is considered in Lauzac et al. [2003]. Bolosky et al. [1997] present a system for serving video data and discuss the schedule-management issues that arise in such a system. The details of a real-time streaming protocol can be found at http://www.rtsp.org. Tudor [1995] gives a tutorial on MPEG-2. A tutorial on video compression techniques can be found at http://www.wave-report.com/tutorials/VC.htm.

Part Eight

Case Studies

We can now integrate the concepts described in this book by describing real operating systems. Two such systems are covered in great detail—Linux and Windows XP. We chose Linux for several reasons: it is popular, it is freely available, and it represents a full-featured UNIX system. This gives a student of operating systems an opportunity to read—and modify—*real* operating-system source code.

We also cover Windows XP in great detail. This recent operating system from Microsoft is gaining popularity, not only in the stand-alone-machine market, but also in the workgroup-server market. We chose Windows XP because it provides an opportunity for us to study a modern operating system that has a design and implementation drastically different from those of UNIX.

In addition, we briefly discuss other highly influential operating systems. We have chosen the order of presentation to highlight the similarities and differences among the systems; it is not strictly chronological and does not reflect the relative importance of the systems.

Finally, we provide on-line coverage of three other systems. The FreeBSD system is another UNIX system. However, whereas Linux combines features from several UNIX systems, FreeBSD is based on the BSD model of UNIX. FreeBSD source code, like Linux source code, is freely available. The Mach operating system is a modern operating system that provides compatibility with BSD UNIX. Windows is another modern operating system from Microsoft for Intel Pentium and later microprocessors; it is compatible with MS-DOS and Microsoft Windows applications.

The Linux System



This chapter presents an in-depth examination of the Linux operating system. By examining a complete, real system, we can see how the concepts we have discussed relate both to one another and to practice.

Linux is a version of UNIX that has gained popularity in recent years. In this chapter, we look at the history and development of Linux and cover the user and programmer interfaces that Linux presents—interfaces that owe a great deal to the UNIX tradition. We also discuss the internal methods by which Linux implements these interfaces. Linux is a rapidly evolving operating system. This chapter describes developments through the Linux 2.6 kernel, which was released in late 2003.

CHAPTER OBJECTIVES

- To explore the history of the UNIX operating system from which Linux is derived and the principles upon which Linux is designed.
- To examine the Linux process model and illustrate how Linux schedules processes and provides interprocess communication.
- To look at memory management in Linux.
- To explore how Linux implements file systems and manages I/O devices.

21.1 Linux History

Linux looks and feels much like any other UNIX system; indeed, UNIX compatibility has been a major design goal of the Linux project. However, Linux is much younger than most UNIX systems. Its development began in 1991, when a Finnish student, Linus Torvalds, wrote and christened **Linux**, a small but self-contained kernel for the 80386 processor, the first true 32-bit processor in Intel's range of PC-compatible CPUs.

Early in its development, the Linux source code was made available free on the Internet. As a result, Linux's history has been one of collaboration by many users from all around the world, corresponding almost exclusively over the Internet. From an initial kernel that partially implemented a small subset of

the UNIX system services, the Linux system has grown to include much UNIX functionality.

In its early days, Linux development revolved largely around the central operating-system kernel—the core, privileged executive that manages all system resources and that interacts directly with the computer hardware. We need much more than this kernel to produce a full operating system, of course. It is useful to make the distinction between the Linux kernel and a Linux system. The **Linux kernel** is an entirely original piece of software developed from scratch by the Linux community. The **Linux system**, as we know it today, includes a multitude of components, some written from scratch, others borrowed from other development projects, and still others created in collaboration with other teams.

The basic Linux system is a standard environment for applications and user programming, but it does not enforce any standard means of managing the available functionality as a whole. As Linux has matured, a need has arisen for another layer of functionality on top of the Linux system. This need has been met by various Linux distributions. A **Linux distribution** includes all the standard components of the Linux system, plus a set of administrative tools to simplify the initial installation and subsequent upgrading of Linux and to manage installation and removal of other packages on the system. A modern distribution also typically includes tools for management of file systems, creation and management of user accounts, administration of networks, web browsers, word processors, and so on.

21.1.1 The Linux Kernel

The first Linux kernel released to the public was Version 0.01, dated May 14, 1991. It had no networking, ran only on 80386-compatible Intel processors and PC hardware, and had extremely limited device-driver support. The virtual memory subsystem was also fairly basic and included no support for memory-mapped files; however, even this early incarnation supported shared pages with copy-on-write. The only file system supported was the Minix file system —the first Linux kernels were cross-developed on a Minix platform. However, the kernel did implement proper UNIX processes with protected address spaces.

The next milestone version, Linux 1.0, was released on March 14, 1994. This release culminated three years of rapid development of the Linux kernel. Perhaps the single biggest new feature was networking: 1.0 included support for UNIX's standard TCP/IP networking protocols, as well as a BSD-compatible socket interface for networking programming. Device-driver support was added for running IP over an Ethernet or (using PPP or SLIP protocols) over serial lines or modems.

The 1.0 kernel also included a new, much enhanced file system without the limitations of the original Minix file system and supported a range of SCSI controllers for high-performance disk access. The developers extended the virtual memory subsystem to support paging to swap files and memory mapping of arbitrary files (but only read-only memory mapping was implemented in 1.0).

A range of extra hardware support was also included in this release. Although still restricted to the Intel PC platform, hardware support had grown to include floppy-disk and CD-ROM devices, as well as sound cards, a range of mice, and international keyboards. Floating-point emulation was provided

in the kernel for 80386 users who had no 80387 math coprocessor; System V UNIX-style **interprocess communication (IPC)**, including shared memory, semaphores, and message queues, was implemented. Simple support for dynamically loadable and unloadable kernel modules was supplied as well.

At this point, development started on the 1.1 kernel stream, but numerous bug-fix patches were released subsequently against 1.0. A pattern was adopted as the standard numbering convention for Linux kernels. Kernels with an odd minor-version number, such as 1.1, 1.3, and 2.1, are **development kernels**; even-numbered minor-version numbers are stable **production kernels**. Updates against the stable kernels are intended only as remedial versions, whereas the development kernels may include newer and relatively untested functionality.

In March 1995, the 1.2 kernel was released. This release did not offer nearly the same improvement in functionality as the 1.0 release, but it did support a much wider variety of hardware, including the new PCI hardware bus architecture. Developers added another PC-specific feature—support for the 80386 CPU's virtual 8086 mode—to allow emulation of the DOS operating system for PC computers. They also updated the networking stack to provide support for the IPX protocol and made the IP implementation more complete by including accounting and firewalling functionality.

The 1.2 kernel was the final PC-only Linux kernel. The source distribution for Linux 1.2 included partially implemented support for SPARC, Alpha, and MIPS CPUs, but full integration of these other architectures did not begin until after the 1.2 stable kernel was released.

The Linux 1.2 release concentrated on wider hardware support and more complete implementations of existing functionality. Much new functionality was under development at the time, but integration of the new code into the main kernel source code had been deferred until after the stable 1.2 kernel had been released. As a result, the 1.3 development stream saw a great deal of new functionality added to the kernel.

This work was finally released as Linux 2.0 in June 1996. This release was given a major version-number increment on account of two major new capabilities: support for multiple architectures, including a fully 64-bit native Alpha port, and support for multiprocessor architectures. Linux distributions based on 2.0 are also available for the Motorola 68000-series processors and for Sun's SPARC systems. A derived version of Linux running on top of the Mach microkernel also runs on PC and PowerMac systems.

The changes in 2.0 did not stop there. The memory-management code was substantially improved to provide a unified cache for file-system data independent of the caching of block devices. As a result of this change, the kernel offered greatly increased file-system and virtual memory performance. For the first time, file-system caching was extended to networked file systems, and writable memory-mapped regions also were supported.

The 2.0 kernel also included much improved TCP/IP performance, and a number of new networking protocols were added, including AppleTalk, AX.25 amateur radio networking, and ISDN support. The ability to mount remote netware and SMB (Microsoft LanManager) network volumes was added.

Other major improvements in 2.0 were support for internal kernel threads, for handling dependencies between loadable modules, and for automatic loading of modules on demand. Dynamic configuration of the kernel at run time was much improved through a new, standardized configuration interface.

Additional new features included file-system quotas and POSIX-compatible real-time process-scheduling classes.

Improvements continued with the release of Linux 2.2 in January 1999. A port for UltraSPARC systems was added. Networking was enhanced with more flexible firewalling, better routing and traffic management, and support for TCP large window and selective acks. Acorn, Apple, and NT disks could now be read, and NFS was enhanced and a kernel-mode NFS daemon added. Signal handling, interrupts, and some I/O were locked at a finer level than before to improve symmetric multiprocessor (SMP) performance.

Advances in the 2.4 and 2.6 releases of the kernel include increased support for SMP systems, journaling file systems, and enhancements to the memory-management system. The process scheduler has been modified in version 2.6, providing an efficient $O(1)$ scheduling algorithm. In addition, the Linux 2.6 kernel is now preemptive, allowing a process to be preempted while running in kernel mode.

21.1.2 The Linux System

In many ways, the Linux kernel forms the core of the Linux project, but other components make up the complete Linux operating system. Whereas the Linux kernel is composed entirely of code written from scratch specifically for the Linux project, much of the supporting software that makes up the Linux system is not exclusive to Linux but is common to a number of UNIX-like operating systems. In particular, Linux uses many tools developed as part of Berkeley's BSD operating system, MIT's X Window System, and the Free Software Foundation's GNU project.

This sharing of tools has worked in both directions. The main system libraries of Linux were originated by the GNU project, but the Linux community greatly improved the libraries by addressing omissions, inefficiencies, and bugs. Other components, such as the **GNU C compiler (gcc)**, were already of sufficiently high quality to be used directly in Linux. The networking-administration tools under Linux were derived from code first developed for 4.3 BSD, but more recent BSD derivatives, such as FreeBSD, have borrowed code from Linux in return. Examples include the Intel floating-point-emulation math library and the PC sound-hardware device drivers.

The Linux system as a whole is maintained by a loose network of developers collaborating over the Internet, with small groups or individuals having responsibility for maintaining the integrity of specific components. A small number of public Internet file-transfer-protocol (ftp) archive sites act as de facto standard repositories for these components. The **File System Hierarchy Standard** document is also maintained by the Linux community as a means of keeping compatibility across the various system components. This standard specifies the overall layout of a standard Linux file system; it determines under which directory names configuration files, libraries, system binaries, and run-time data files should be stored.

21.1.3 Linux Distributions

In theory, anybody can install a Linux system by fetching the latest revisions of the necessary system components from the ftp sites and compiling them. In Linux's early days, this operation was often precisely what a Linux user

had to carry out. As Linux has matured, however, various individuals and groups have attempted to make this job less painful by providing a standard, precompiled set of packages for easy installation.

These collections, or distributions, include much more than just the basic Linux system. They typically include extra system-installation and management utilities, as well as precompiled and ready-to-install packages of many of the common UNIX tools, such as news servers, web browsers, text-processing and editing tools, and even games.

The first distributions managed these packages by simply providing a means of unpacking all the files into the appropriate places. One of the important contributions of modern distributions, however, is advanced package management. Today's Linux distributions include a package-tracking database that allows packages to be installed, upgraded, or removed painlessly.

The SLS distribution, dating back to the early days of Linux, was the first collection of Linux packages that was recognizable as a complete distribution. Although it could be installed as a single entity, SLS lacked the package-management tools now expected of Linux distributions. The **Slackware** distribution represented a great improvement in overall quality, even though it also had poor package management; it is still one of the most widely installed distributions in the Linux community.

Since Slackware's release, many commercial and noncommercial Linux distributions have become available. **Red Hat** and **Debian** are particularly popular distributions; the first comes from a commercial Linux support company and the second from the free-software Linux community. Other commercially supported versions of Linux include distributions from **Caldera**, **Craftworks**, and **WorkGroup Solutions**. A large Linux following in Germany has resulted in several dedicated German-language distributions, including versions from **SuSE** and **Unifix**. There are too many Linux distributions in circulation for us to list all of them here. The variety of distributions does not prohibit compatibility across Linux distributions, however. The RPM package file format is used, or at least understood, by the majority of distributions, and commercial applications distributed in this format can be installed and run on any distribution that can accept RPM files.

21.1.4 Linux Licensing

The Linux kernel is distributed under the GNU general public license (GPL), the terms of which are set out by the Free Software Foundation. Linux is not public-domain software. **Public domain** implies that the authors have waived copyright rights in the software, but copyright rights in Linux code are still held by the code's various authors. Linux is *free* software, however, in the sense that people can copy it, modify it, use it in any manner they want, and give away their own copies, without any restrictions.

The main implications of Linux's licensing terms are that nobody using Linux, or creating her own derivative of Linux (a legitimate exercise), can make the derived product proprietary. Software released under the GPL cannot be redistributed as a binary-only product. If you release software that includes any components covered by the GPL, then, under the GPL, you must make source code available alongside any binary distributions. (This restriction does

not prohibit making—or even selling—binary-only software distributions, as long as anybody who receives binaries is also given the opportunity to get source code, for a reasonable distribution charge.)

21.2 Design Principles

In its overall design, Linux resembles any other traditional, nonmicrokernel UNIX implementation. It is a multiuser, multitasking system with a full set of UNIX-compatible tools. Linux's file system adheres to traditional UNIX semantics, and the standard UNIX networking model is implemented fully. The internal details of Linux's design have been influenced heavily by the history of this operating system's development.

Although Linux runs on a wide variety of platforms, it was developed exclusively on PC architecture. A great deal of that early development was carried out by individual enthusiasts, rather than by well-funded development or research facilities, so from the start Linux attempted to squeeze as much functionality as possible from limited resources. Today, Linux can run happily on a multiprocessor machine with hundreds of megabytes of main memory and many gigabytes of disk space, but it is still capable of operating usefully in under 4 MB of RAM.

As PCs became more powerful and as memory and hard disks became cheaper, the original, minimalist Linux kernels grew to implement more UNIX functionality. Speed and efficiency are still important design goals, but much of the recent and current work on Linux has concentrated on a third major design goal: standardization. One of the prices paid for the diversity of UNIX implementations currently available is that source code written for one flavor may not necessarily compile or run correctly on another. Even when the same system calls are present on two different UNIX systems, they do not necessarily behave in exactly the same way. The POSIX standards comprise a set of specifications of different aspects of operating-system behavior. There are POSIX documents for common operating-system functionality and for extensions such as process threads and real-time operations. Linux is designed to be compliant with the relevant POSIX documents; at least two Linux distributions have achieved official POSIX certification.

Because it presents standard interfaces to both the programmer and the user, Linux presents few surprises to anybody familiar with UNIX. We do not detail these interfaces here. The sections on the programmer interface (Section A.3) and user interface (Section A.4) of BSD apply equally well to Linux. By default, however, the Linux programming interface adheres to SVR4 UNIX semantics, rather than to BSD behavior. A separate set of libraries is available to implement BSD semantics in places where the two behaviors are significantly different.

Many other standards exist in the UNIX world, but full certification of Linux against them is sometimes slowed because they are often available only for a fee, and the expense involved in certifying an operating system's compliance with most standards is substantial. However, supporting a wide base of applications is important for any operating system, so implementation of standards is a major goal for Linux development, even if the implementation is not formally certified. In addition to the basic POSIX standard, Linux currently

supports the POSIX threading extensions—Pthreads—and a subset of the POSIX extensions for real-time process control.

21.2.1 Components of a Linux System

The Linux system is composed of three main bodies of code, in line with most traditional UNIX implementations:

1. **Kernel**. The kernel is responsible for maintaining all the important abstractions of the operating system, including such things as virtual memory and processes.
2. **System libraries**. The system libraries define a standard set of functions through which applications can interact with the kernel. These functions implement much of the operating-system functionality that does not need the full privileges of kernel code.
3. **System utilities**. The system utilities are programs that perform individual, specialized management tasks. Some system utilities may be invoked just once to initialize and configure some aspect of the system; others—known as *daemons* in UNIX terminology—may run permanently, handling such tasks as responding to incoming network connections, accepting logon requests from terminals, and updating log files.

Figure 21.1 illustrates the various components that make up a full Linux system. The most important distinction here is between the kernel and everything else. All the kernel code executes in the processor's privileged mode with full access to all the physical resources of the computer. Linux refers to this privileged mode as **kernel mode**. Under Linux, no user-mode code is built into the kernel. Any operating-system-support code that does not need to run in kernel mode is placed into the system libraries instead.

Although various modern operating systems have adopted a message-passing architecture for their kernel internals, Linux retains UNIX's historical model: The kernel is created as a single, monolithic binary. The main reason is to improve performance: Because all kernel code and data structures are kept in a single address space, no context switches are necessary when a process calls an operating-system function or when a hardware interrupt is delivered. Not

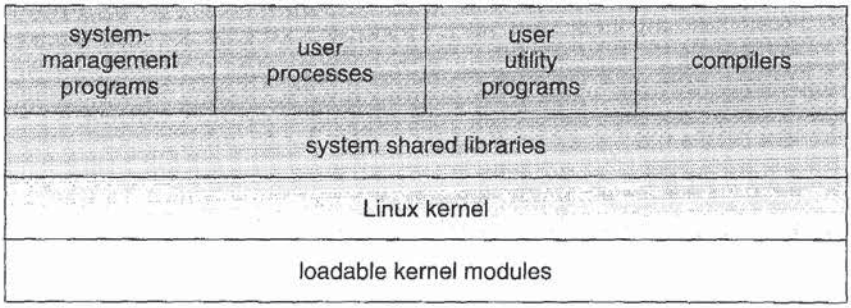


Figure 21.1 Components of the Linux system.

only the core scheduling and virtual memory code occupies this address space; *all* kernel code, including all device drivers, file systems, and networking code, is present in the same single address space.

Even though all the kernel components share this same melting pot, there is still room for modularity. In the same way that user applications can load shared libraries at run time to pull in a needed piece of code, so the Linux kernel can load (and unload) modules dynamically at run time. The kernel does not necessarily need to know in advance which modules may be loaded —they are truly independent loadable components.

The Linux kernel forms the core of the Linux operating system. It provides all the functionality necessary to run processes, and it provides system services to give arbitrated and protected access to hardware resources. The kernel implements all the features required to qualify as an operating system. On its own, however, the operating system provided by the Linux kernel looks nothing like a UNIX system. It is missing many of the extra features of UNIX, and the features that it does provide are not necessarily in the format in which a UNIX application expects them to appear. The operating-system interface visible to running applications is not maintained directly by the kernel. Rather, applications make calls to the system libraries, which in turn call the operating-system services as necessary.

The system libraries provide many types of functionality. At the simplest level, they allow applications to make kernel-system-service requests. Making a system call involves transferring control from unprivileged user mode to privileged kernel mode; the details of this transfer vary from architecture to architecture. The libraries take care of collecting the system-call arguments and, if necessary, arranging those arguments in the special form necessary to make the system call.

The libraries may also provide more complex versions of the basic system calls. For example, the C language's buffered file-handling functions are all implemented in the system libraries, providing more advanced control of file I/O than the basic kernel system calls. The libraries also provide routines that do not correspond to system calls at all, such as sorting algorithms, mathematical functions, and string-manipulation routines. All the functions necessary to support the running of UNIX or POSIX applications are implemented here in the system libraries.

The Linux system includes a wide variety of user-mode programs—both system utilities and user utilities. The system utilities include all the programs necessary to initialize the system, such as those to configure network devices and to load kernel modules. Continually running server programs also count as system utilities; such programs handle user login requests, incoming network connections, and the printer queues.

Not all the standard utilities serve key system-administration functions. The UNIX user environment contains a large number of standard utilities to do simple everyday tasks, such as listing directories, moving and deleting files, and displaying the contents of a file. More complex utilities can perform text-processing functions, such as sorting textual data and performing pattern searches on input text. Together, these utilities form a standard tool set that users can expect on any UNIX system; although they do not perform any operating-system function, they are an important part of the basic Linux system.

21.3 Kernel Modules

The Linux kernel has the ability to load and unload arbitrary sections of kernel code on demand. These loadable kernel modules run in privileged kernel mode and as a consequence have full access to all the hardware capabilities of the machine on which they run. In theory, there is no restriction on what a kernel module is allowed to do; typically, a module might implement a device driver, a file system, or a networking protocol.

Kernel modules are convenient for several reasons. Linux's source code is free, so anybody wanting to write kernel code is able to compile a modified kernel and to reboot to load that new functionality; however, recompiling, relinking, and reloading the entire kernel is a cumbersome cycle to undertake when you are developing a new driver. If you use kernel modules, you do not have to make a new kernel to test a new driver—the driver can be compiled on its own and loaded into the already-running kernel. Of course, once a new driver is written, it can be distributed as a module so that other users can benefit from it without having to rebuild their kernels.

This latter point has another implication. Because it is covered by the GPL license, the Linux kernel cannot be released with proprietary components added to it, unless those new components are also released under the GPL and the source code for them is made available on demand. The kernel's module interface allows third parties to write and distribute, on their own terms, device drivers or file systems that could not be distributed under the GPL.

Kernel modules allow a Linux system to be set up with a standard, minimal kernel, without any extra device drivers built in. Any device drivers that the user needs can be either loaded explicitly by the system at startup or loaded automatically by the system on demand and unloaded when not in use. For example, a CD-ROM driver might be loaded when a CD is mounted and unloaded from memory when the CD is dismounted from the file system.

The module support under Linux has three components:

1. The **module management** allows modules to be loaded into memory and to talk to the rest of the kernel.
2. The **driver registration** allows modules to tell the rest of the kernel that a new driver has become available.
3. A **conflict-resolution mechanism** allows different device drivers to reserve hardware resources and to protect those resources from accidental use by another driver.

21.3.1 Module Management

Loading a module requires more than just loading its binary contents into kernel memory. The system must also make sure that any references the module makes to kernel symbols or entry points are updated to point to the correct locations in the kernel's address space. Linux deals with this reference updating by splitting the job of module loading into two separate sections: the management of sections of module code in kernel memory and the handling of symbols that modules are allowed to reference.

Linux maintains an internal symbol table in the kernel. This symbol table does not contain the full set of symbols defined in the kernel during the latter's compilation; rather, a symbol must be exported explicitly by the kernel. The set of exported symbols constitutes a well-defined interface by which a module can interact with the kernel.

Although exporting symbols from a kernel function requires an explicit request by the programmer, no special effort is needed to import those symbols into a module. A module writer just uses the standard external linking of the C language: Any external symbols referenced by the module but not declared by it are simply marked as unresolved in the final module binary produced by the compiler. When a module is to be loaded into the kernel, a system utility first scans the module for these unresolved references. All symbols that still need to be resolved are looked up in the kernel's symbol table, and the correct addresses of those symbols in the currently running kernel are substituted into the module's code. Only then is the module passed to the kernel for loading. If the system utility cannot resolve any references in the module by looking them up in the kernel's symbol table, then the module is rejected.

The loading of the module is performed in two stages. First, the module-loader utility asks the kernel to reserve a continuous area of virtual kernel memory for the module. The kernel returns the address of the memory allocated, and the loader utility can use this address to relocate the module's machine code to the correct loading address. A second system call then passes the module, plus any symbol table that the new module wants to export, to the kernel. The module itself is now copied verbatim into the previously allocated space, and the kernel's symbol table is updated with the new symbols for possible use by other modules not yet loaded.

The final module-management component is the module requestor. The kernel defines a communication interface to which a module-management program can connect. With this connection established, the kernel will inform the management process whenever a process requests a device driver, file system, or network service that is not currently loaded and will give the manager the opportunity to load that service. The original service request will complete once the module is loaded. The manager process regularly queries the kernel to see whether a dynamically loaded module is still in use and unloads that module when it is no longer actively needed.

21.3.2 Driver Registration

Once a module is loaded, it remains no more than an isolated region of memory until it lets the rest of the kernel know what new functionality it provides. The kernel maintains dynamic tables of all known drivers and provides a set of routines to allow drivers to be added to or removed from these tables at any time. The kernel makes sure that it calls a module's startup routine when that module is loaded and calls the module's cleanup routine before that module is unloaded: These routines are responsible for registering the module's functionality.

A module may register many types of drivers and may register more than one driver if it wishes. For example, a device driver might want to register two separate mechanisms for accessing the device. Registration tables include the following items:

- **Device drivers.** These drivers include character devices (such as printers, terminals, and mice), block devices (including all disk drives), and network interface devices.
- **File systems.** The file system may be anything that implements Linux's virtual-file-system calling routines. It might implement a format for storing files on a disk, but it might equally well be a network file system, such as NFS, or a virtual file system whose contents are generated on demand, such as Linux's `/proc` file system.
- **Network protocols.** A module may implement an entire networking protocol, such as IPX, or simply a new set of packet-filtering rules for a network firewall.
- **Binary format.** This format specifies a way of recognizing, and loading, a new type of executable file.

In addition, a module can register a new set of entries in the *sysctl* and */proc* tables, to allow that module to be configured dynamically (Section 21.7.4).

21.3.3 Conflict Resolution

Commercial UNIX implementations are usually sold to run on a vendor's own hardware. One advantage of a single-supplier solution is that the software vendor has a good idea about what hardware configurations are possible. IBM PC hardware, however, comes in a vast number of configurations, with large numbers of possible drivers for devices such as network cards, SCSI controllers, and video display adapters. The problem of managing the hardware configuration becomes more severe when modular device drivers are supported, since the currently active set of devices becomes dynamically variable.

Linux provides a central conflict-resolution mechanism to help arbitrate access to certain hardware resources. Its aims are as follows:

- To prevent modules from clashing over access to hardware resources
- To prevent **autoprobes**—device-driver probes that auto-detect device configuration—from interfering with existing device drivers
- To resolve conflicts among multiple drivers trying to access the same hardware—for example, as when both the parallel printer driver and the parallel-line IP (PLIP) network driver try to talk to the parallel printer port

To these ends, the kernel maintains lists of allocated hardware resources. The PC has a limited number of possible I/O ports (addresses in its hardware I/O address space), interrupt lines, and DMA channels; when any device driver wants to access such a resource, it is expected to reserve the resource with the kernel database first. This requirement incidentally allows the system administrator to determine exactly which resources have been allocated by which driver at any given point.

A module is expected to use this mechanism to reserve in advance any hardware resources that it expects to use. If the reservation is rejected because the resource is not present or is already in use, then it is up to the module

to decide how to proceed. It may fail its initialization and request that it be unloaded if it cannot continue, or it may carry on, using alternative hardware resources.

21.4 Process Management

A process is the basic context within which all user-requested activity is serviced within the operating system. To be compatible with other UNIX systems, Linux must use a process model similar to those of other versions of UNIX. Linux operates differently from UNIX in a few key places, however. In this section, we review the traditional UNIX process model from Section A.3.2 and introduce Linux's own threading model.

21.4.1 The fork() and exec() Process Model

The basic principle of UNIX process management is to separate two operations: the creation of a process and the running of a new program. A new process is created by the `fork()` system call, and a new program is run after a call to `exec()`. These are two distinctly separate functions. A new process may be created with `fork()` without a new program being run—the new subprocess simply continues to execute exactly the same program that the first, parent process was running. Equally, running a new program does not require that a new process be created first: Any process may call `exec()` at any time. The currently running program is immediately terminated, and the new program starts executing in the context of the existing process.

This model has the advantage of great simplicity. Rather than having to specify every detail of the environment of a new program in the system call that runs that program, new programs simply run in their existing environment. If a parent process wishes to modify the environment in which a new program is to be run, it can fork and then, still running the original program in a child process, make any system calls it requires to modify that child process before finally executing the new program.

Under UNIX, then, a process encompasses all the information that the operating system must maintain to track the context of a single execution of a single program. Under Linux, we can break down this context into a number of specific sections. Broadly, process properties fall into three groups: the process identity, environment, and context.

21.4.1.1 Process Identity

A process identity consists mainly of the following items:

- **Process ID (PID).** Each process has a unique identifier. PIDs are used to specify processes to the operating system when an application makes a system call to signal, modify, or wait for another process. Additional identifiers associate the process with a process group (typically, a tree of processes `fork`ed by a single user command) and login session.
- **Credentials.** Each process must have an associated user ID and one or more group IDs (user groups are discussed in Section 10.6.2) that determine the rights of a process to access system resources and files.

- **Personality**. Process personalities are not traditionally found on UNIX systems, but under Linux each process has an associated personality identifier that can modify slightly the semantics of certain system calls. Personalities are primarily used by emulation libraries to request that system calls be compatible with certain flavors of UNIX.

Most of these identifiers are under limited control of the process itself. The process group and session identifiers can be changed if the process wants to start a new group or session. Its credentials can be changed, subject to appropriate security checks. However, the primary PID of a process is unchangeable and uniquely identifies that process until termination.

21.4.1.2 Process Environment

A process's environment is inherited from its parent and is composed of two null-terminated vectors: the argument vector and the environment vector. The **argument vector** simply lists the command-line arguments used to invoke the running program; it conventionally starts with the name of the program itself. The **environment vector** is a list of "NAME=VALUE" pairs that associates named environment variables with arbitrary textual values. The environment is not held in kernel memory but is stored in the process's own user-mode address space as the first datum at the top of the process's stack.

The argument and environment vectors are not altered when a new process is created: The new child process will inherit the environment that its parent possesses. However, a completely new environment is set up when a new program is invoked. On calling `exec()`, a process must supply the environment for the new program. The kernel passes these environment variables to the next program, replacing the process's current environment. The kernel otherwise leaves the environment and command-line vectors alone—their interpretation is left entirely to the user-mode libraries and applications.

The passing of environment variables from one process to the next and the inheriting of these variables by the children of a process provide flexible ways to pass information to components of the user-mode system software. Various important environment variables have conventional meanings to related parts of the system software. For example, the `TERM` variable is set up to name the type of terminal connected to a user's login session; many programs use this variable to determine how to perform operations on the user's display, such as moving the cursor and scrolling a region of text. Programs with multilingual support use the `LANG` variable to determine in which language to display system messages for programs that include multilingual support.

The environment-variable mechanism custom tailors the operating system on a per-process basis, rather than for the system as a whole. Users can choose their own languages or select their own editors independently of one another.

21.4.1.3 Process Context

The process identity and environment properties are usually set up when a process is created and not changed until that process exits. A process may choose to change some aspects of its identity if it needs to do so, or it may alter its environment. In contrast, process context is the state of the running program at any one time; it changes constantly. Process context includes the following parts.

- **Scheduling context**. The most important part of the process context is its scheduling context—the information that the scheduler needs to suspend and restart the process. This information includes saved copies of all the process's registers. Floating-point registers are stored separately and are restored only when needed, so that processes that do not use floating-point arithmetic do not incur the overhead of saving that state. The scheduling context also includes information about scheduling priority and about any outstanding signals waiting to be delivered to the process. A key part of the scheduling context is the process's kernel stack, a separate area of kernel memory reserved for use exclusively by kernel-mode code. Both system calls and interrupts that occur while the process is executing will use this stack.
- **Accounting**. The kernel maintains information about the resources currently being consumed by each process and the total resources consumed by the process in its entire lifetime so far.
- **File table**. The file table is an array of pointers to kernel file structures. When making file-I/O system calls, processes refer to files by their index into this table.
- **File-system context**. Whereas the file table lists the existing open files, the file-system context applies to requests to open new files. The current root and default directories to be used for new file searches are stored here.
- **Signal-handler table**. UNIX systems can deliver asynchronous signals to a process in response to various external events. The signal-handler table defines the routine in the process's address space to be called when specific signals arrive.
- **Virtual memory context**. The virtual memory context describes the full contents of a process's private address space; we discuss it in Section 21.6.

21.4.2 Processes and Threads

Linux provides the `fork()` system call with the traditional functionality of duplicating a process. Linux also provides the ability to create threads using the `clone()` system call. However, Linux does not distinguish between processes and threads. In fact, Linux generally uses the term *task*—rather than *process* or *thread*—when referring to a flow of control within a program. When `clone()` is invoked, it is passed a set of flags that determine how much sharing is to take place between the parent and child tasks. Some of these flags are listed below:

flag	meaning
`CLONE_FS`	File-system information is shared.
`CLONE_VM`	The same memory space is shared.
`CLONE_SIGHAND`	Signal handlers are shared.
`CLONE_FILES`	The set of open files is shared.

Thus, if `clone()` is passed the flags `CLONE_FS`, `CLONE_VM`, `CLONE_SIGHAND`, and `CLONE_FILES`, the parent and child tasks will share the same file-system information (such as the current working directory), the same memory space, the same signal handlers, and the same set of open files. Using `clone()` in this fashion is equivalent to creating a thread in other systems, since the parent task shares most of its resources with its child task. However, if none of these flags is set when `clone()` is invoked, no sharing takes place, resulting in functionality similar to the `fork()` system call.

The lack of distinction between processes and threads is possible because Linux does not hold a process's entire context within the main process data structure; rather, it holds the context within independent subcontexts. Thus, a process's file-system context, file-descriptor table, signal-handler table, and virtual memory context are held in separate data structures. The process data structure simply contains pointers to these other structures, so any number of processes can easily share a subcontext by pointing to the same subcontext as appropriate.

The arguments to the `clone()` system call tell it which subcontexts to copy, and which to share, when it creates a new process. The new process always is given a new identity and a new scheduling context; according to the arguments passed, however, it may either create new subcontext data structures initialized to be copies of the parent's or set up the new process to use the same subcontext data structures being used by the parent. The `fork()` system call is nothing more than a special case of `clone()` that copies all subcontexts, sharing none.

21.5 Scheduling

Scheduling is the job of allocating CPU time to different tasks within an operating system. Normally, we think of scheduling as being the running and interrupting of processes, but another aspect of scheduling is also important to Linux: the running of the various kernel tasks. Kernel tasks encompass both tasks that are requested by a running process and tasks that execute internally on behalf of a device driver.

21.5.1 Process Scheduling

Linux has two separate process-scheduling algorithms. One is a time-sharing algorithm for fair, preemptive scheduling among multiple processes; the other is designed for real-time tasks, where absolute priorities are more important than fairness.

The scheduling algorithm used for routine, time-sharing tasks received a major overhaul with version 2.5 of the kernel. Prior to version 2.5, the Linux kernel ran a variation of the traditional UNIX scheduling algorithm. Among other issues, problems with the traditional UNIX scheduler are that it does not provide adequate support for SMP systems and that it does not scale well as the number of tasks on the system grows. The overhaul of the scheduler with version 2.5 of the kernel now provides a scheduling algorithm that runs in constant time—known as $O(1)$—regardless of the number of tasks on the system. The new scheduler also provides increased support for SMP, including

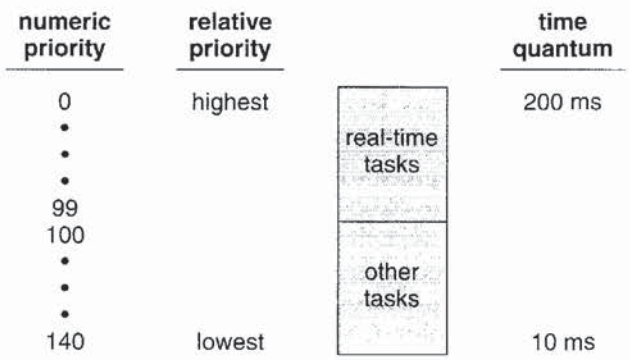


Figure 21.2 The relationship between priorities and time-slice length.

processor affinity and load balancing, as well as maintaining fairness and support for interactive tasks.

The Linux scheduler is a preemptive, priority-based algorithm with two separate priority ranges: a **real-time** range from 0 to 99 and a **nice** value ranging from 100 to 140. These two ranges map into a global priority scheme whereby numerically lower values indicate higher priorities.

Unlike schedulers for many other systems, Linux assigns higher-priority tasks longer time quanta and vice-versa. Because of the unique nature of the scheduler, this is appropriate for Linux, as we shall soon see. The relationship between priorities and time-slice length is shown in Figure 21.2.

A runnable task is considered eligible for execution on the CPU so long as it has time remaining in its time slice. When a task has exhausted its time slice, it is considered **expired** and is not eligible for execution again until all other tasks have also exhausted their time quanta. The kernel maintains a list of all runnable tasks in a **runqueue** data structure. Because of its support for SMP, each processor maintains its own runqueue and schedules itself independently. Each runqueue contains two priority arrays—**active** and **expired**. The active array contains all tasks with time remaining in their time slices, and the expired array contains all expired tasks. Each of these priority arrays includes a list of tasks indexed according to priority (Figure 21.3). The scheduler chooses the task with the highest priority from the active array for execution on the CPU. On multiprocessor machines, this means that each processor is scheduling the highest-priority task from its own runqueue structure. When all tasks have exhausted their time slices (that is, the active array is empty), the two priority arrays are exchanged as the expired array becomes the active array and vice-versa.

Tasks are assigned dynamic priorities that are based on the *nice* value plus or minus up to the value 5 based upon the interactivity of the task. Whether a value is added to or subtracted from a task's *nice* value depends on the interactivity of the task. A task's interactivity is determined by how long it has been sleeping while waiting for I/O. Tasks that are more interactive typically have longer sleep times and therefore are more likely to have an adjustment closer to -5, as the scheduler favors such interactive tasks. Conversely, tasks with shorter sleep times are often more CPU-bound and thus will have their priorities lowered.



Figure 21.3 List of tasks indexed according to priority.

The recalculation of a task's dynamic priority occurs when the task has exhausted its time quantum and is to be moved to the expired array. Thus, when the two arrays are exchanged, all tasks in the new active array have been assigned new priorities and corresponding time slices.

Linux's real-time scheduling is simpler still. Linux implements the two real-time scheduling classes required by POSIX.1b: first-come, first-served (FCFS) and round-robin (Sections 5.3.1 and 5.3.4, respectively). In both cases, each process has a priority in addition to its scheduling class. Processes of different priorities can compete with one another to some extent in time-sharing scheduling; in real-time scheduling, however, the scheduler always runs the process with the highest priority. Among processes of equal priority, it runs the process that has been waiting longest. The only difference between FCFS and round-robin scheduling is that FCFS processes continue to run until they either exit or block, whereas a round-robin process will be preempted after a while and will be moved to the end of the scheduling queue, so round-robin processes of equal priority will automatically time-share among themselves. Unlike routine time-sharing tasks, real-time tasks are assigned static priorities.

Linux's real-time scheduling is soft—rather than hard—real time. The scheduler offers strict guarantees about the relative priorities of real-time processes, but the kernel does not offer any guarantees about how quickly a real-time process will be scheduled once that process becomes runnable.

21.5.2 Kernel Synchronization

The way the kernel schedules its own operations is fundamentally different from the way it schedules processes. A request for kernel-mode execution can occur in two ways. A running program may request an operating-system service, either explicitly via a system call or implicitly—for example, when a page fault occurs. Alternatively, a device driver may deliver a hardware interrupt that causes the CPU to start executing a kernel-defined handler for that interrupt.

The problem posed to the kernel is that all these tasks may try to access the same internal data structures. If one kernel task is in the middle of accessing some data structure when an interrupt service routine executes, then that service routine cannot access or modify the same data without risking data corruption. This fact relates to the idea of critical sections—portions of code that access shared data and that must not be allowed to execute concurrently. As a result, kernel synchronization involves much more than just process

scheduling. A framework is required that allows kernel tasks to run without violating the integrity of shared data.

Prior to version 2.6, Linux was a nonpreemptive kernel, meaning that a process running in kernel mode could not be preempted—even if a higher-priority process became available to run. With version 2.6, the Linux kernel became fully preemptive; so a task can now be preempted when it is running in the kernel.

The Linux kernel provides spinlocks and semaphores (as well as reader–writer versions of these two locks) for locking in the kernel. On SMP machines, the fundamental locking mechanism is a spinlock; the kernel is designed so that the spinlock is held only for short durations. On single-processor machines, spinlocks are inappropriate for use and are replaced by enabling and disabling kernel preemption. That is, on single-processor machines, rather than holding a spinlock, the task disables kernel preemption. When the task would otherwise release the spinlock, it enables kernel preemption. This pattern is summarized below:

single processor	multiple processors
Disable kernel preemption.	Acquire spin lock.
Enable kernel preemption.	Release spin lock.

Linux uses an interesting approach to disable and enable kernel preemption. It provides two simple system calls—`preempt_disable()` and `preempt_enable()`—for disabling and enabling kernel preemption. However, in addition, the kernel is not preemptible if a kernel-mode task is holding a lock. To enforce this rule, each task in the system has a `thread-info` structure that includes the field `preempt_count`, which is a counter indicating the number of locks being held by the task. When a lock is acquired, `preempt_count` is incremented. Likewise, it is decremented when a lock is released. If the value of `preempt_count` for the task currently running is greater than zero, it is not safe to preempt the kernel, as this task currently holds a lock. If the count is zero, the kernel can safely be interrupted, assuming there are no outstanding calls to `preempt_disable()`.

Spinlocks—along with enabling and disabling kernel preemption—are used in the kernel only when the lock is held for short durations. When a lock must be held for longer periods, semaphores are used.

The second protection technique that Linux uses applies to critical sections that occur in interrupt service routines. The basic tool is the processor's interrupt-control hardware. By disabling interrupts (or using spinlocks) during a critical section, the kernel guarantees that it can proceed without the risk of concurrent access of shared data structures.

However, there is a penalty for disabling interrupts. On most hardware architectures, interrupt enable and disable instructions are expensive. Furthermore, as long as interrupts remain disabled, all I/O is suspended, and any device waiting for servicing will have to wait until interrupts are reenabled; so performance degrades. The Linux kernel uses a synchronization architecture that allows long critical sections to run for their entire duration without having interrupts disabled. This ability is especially useful in the networking code: An

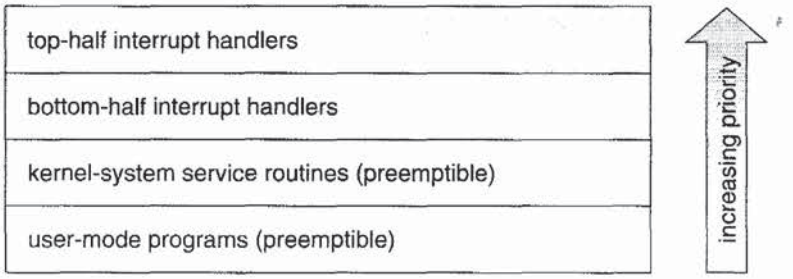


Figure 21.4 Interrupt protection levels.

interrupt in a network device driver can signal the arrival of an entire network packet, which may result in a great deal of code being executed to disassemble, route, and forward that packet within the interrupt service routine.

Linux implements this architecture by separating interrupt service routines into two sections: the top half and the bottom half. The **top half** is a normal interrupt service routine and runs with recursive interrupts disabled; interrupts of a higher priority may interrupt the routine, but interrupts of the same or lower priority are disabled. The **bottom half** of a service routine is run, with all interrupts enabled, by a miniature scheduler that ensures that bottom halves never interrupt themselves. The bottom-half scheduler is invoked automatically whenever an interrupt service routine exits.

This separation means that the kernel can complete any complex processing that has to be done in response to an interrupt without worrying about being interrupted itself. If another interrupt occurs while a bottom half is executing, then that interrupt can request that the same bottom half execute, but the execution will be deferred until the one currently running completes. Each execution of the bottom half can be interrupted by a top half but can never be interrupted by a similar bottom half.

The top-half/bottom-half architecture is completed by a mechanism for disabling selected bottom halves while executing normal, foreground kernel code. The kernel can code critical sections easily using this system. Interrupt handlers can code their critical sections as bottom halves; and when the foreground kernel wants to enter a critical section, it can disable any relevant bottom halves to prevent any other critical sections from interrupting it. At the end of the critical section, the kernel can reenable the bottom halves and run any bottom-half tasks that have been queued by top-half interrupt service routines during the critical section.

Figure 21.4 summarizes the various levels of interrupt protection within the kernel. Each level may be interrupted by code running at a higher level but will never be interrupted by code running at the same or a lower level; except for user-mode code, user processes can always be preempted by another process when a time-sharing scheduling interrupt occurs.

21.5.3 Symmetric Multiprocessing

The Linux 2.0 kernel was the first stable Linux kernel to support **symmetric multiprocessor (SMP)** hardware, allowing separate processes to execute in parallel on separate processors. Originally, the implementation of SMP imposed

the restriction that only one processor at a time could be executing kernel-mode code.

In version 2.2 of the kernel, a single kernel spinlock (sometimes termed **BKL** for "big kernel lock") was created to allow multiple processes (running on different processors) to be active in the kernel concurrently. However, the BKL provided a very coarse level of locking granularity. Later releases of the kernel made the SMP implementation more scalable by splitting this single kernel spinlock into multiple locks, each of which protects only a small subset of the kernel's data structures. Such spinlocks are described in Section 21.5.2. The 2.6 kernel provided additional SMP enhancements, including processor affinity and load-balancing algorithms.

21.6 Memory Management

Memory management under Linux has two components. The first deals with allocating and freeing physical memory—pages, groups of pages, and small blocks of memory. The second handles virtual memory, which is memory mapped into the address space of running processes. In this section, we describe these two components and then examine the mechanisms by which the loadable components of a new program are brought into a process's virtual memory in response to an `exec()` system call.

21.6.1 Management of Physical Memory

Due to specific hardware characteristics, Linux separates physical memory into three different **zones** identifying different regions of memory. The zones are identified as:

- `ZONE_DMA`
- `ZONE_NORMAL`
- `ZONE_HIGHMEM`

These zones are architecture specific. For example, on the Intel 80x86 architecture, certain ISA (industry standard architecture) devices can only access the lower 16 MB of physical memory using DMA. On these systems, the first 16 MB of physical memory comprise `ZONE_DMA`. `ZONE_NORMAL` identifies physical memory that is mapped to the CPU's address space. This zone is used for most routine memory requests. For architectures that do not limit what DMA can access, `ZONE_DMA` is not present, and `ZONE_NORMAL` is used. Finally, `ZONE_HIGHMEM` (for "high memory") refers to physical memory that is not mapped into the kernel address space. For example, on the 32-bit Intel architecture (where 2^{32} provides a 4-GB address space), the kernel is mapped into the first 896 MB of the address space; the remaining memory is referred to as **high memory** and is allocated from `ZONE_HIGHMEM`. The relationship of zones and physical addresses on the Intel 80x86 architecture is shown in Figure 21.5. The kernel maintains a list of free pages for each zone. When a request for physical memory arrives, the kernel satisfies the request using the appropriate zone.

zone	physical memory
ZONE_DMA	< 16 MB
ZONE_NORMAL	16 .. 896 MB
ZONE_HIGHMEM	> 896 MB

Figure 21.5 Relationship of zones and physical addresses on the Intel 80x86.

The primary physical-memory manager in the Linux kernel is the **page allocator**. Each zone has its own allocator, which is responsible for allocating and freeing all physical pages for the zone, and it is capable of allocating ranges of physically contiguous pages on request. The allocator uses a **buddy system** (Section 9.8.1) to keep track of available physical pages. In this scheme, adjacent units of allocatable memory are paired together (hence its name). Each allocatable memory region has an adjacent partner (or buddy). Whenever two allocated partner regions are freed up, they are combined to form a larger region—a *buddy heap*. That larger region also has a partner, with which it can combine to form a still larger free region. Conversely, if a small memory request cannot be satisfied by allocation of an existing small free region, then a larger free region will be subdivided into two partners to satisfy the request. Separate linked lists are used to record the free memory regions of each allowable size; under Linux, the smallest size allocatable under this mechanism is a single physical page. Figure 21.6 shows an example of buddy-heap allocation. A 4-KB region is being allocated, but the smallest available region is 16 KB. The region is broken up recursively until a piece of the desired size is available.

Ultimately, all memory allocations in the Linux kernel are made either statically, by drivers that reserve a contiguous area of memory during system boot time, or dynamically, by the page allocator. However, kernel functions do not have to use the basic allocator to reserve memory. Several specialized memory-management subsystems use the underlying page allocator to manage their own pools of memory. The most important are the virtual memory system, described in Section 21.6.2; the `kmalloc()` variable-length allocator;

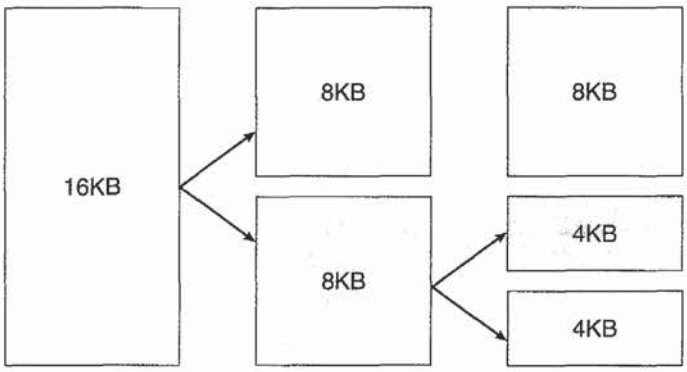


Figure 21.6 Splitting of memory in the buddy system.

the slab allocator, used for allocating memory for kernel data structures; and the page cache, used for caching pages belonging to files.

Many components of the Linux operating system need to allocate entire pages on request, but often smaller blocks of memory are required. The kernel provides an additional allocator for arbitrary-sized requests, where the size of a request is not known in advance and may be only a few bytes, rather than an entire page. Analogous to the C language's `malloc()` function, this `kmalloc()` service allocates entire pages on demand but then splits them into smaller pieces. The kernel maintains a set of lists of pages in use by the `kmalloc()` service. Allocating memory involves working out the appropriate list and either taking the first free piece available on the list or allocating a new page and splitting it up. Memory regions claimed by the `kmalloc()` system are allocated permanently until they are freed explicitly; the `kmalloc()` system cannot relocate or reclaim these regions in response to memory shortages.

Another strategy adopted by Linux for allocating kernel memory is known as slab allocation. A **slab** is used for allocating memory for kernel data structures and is made up of one or more physically contiguous pages. A **cache** consists of one or more slabs and there is a single cache for each unique kernel data structure —for example, a cache for the data structure representing process descriptors, a cache for file objects, a cache for semaphores, and so forth. Each cache is populated with **objects** that are instantiations of the kernel data structure the cache represents. For example, the cache representing semaphores stores instances of semaphore objects, the cache representing process descriptors stores instances of process descriptor objects, etc. The relationship among slabs, caches, and objects is shown in Figure 21.7. The figure shows two kernel objects 3 KB in size and three objects 7 KB in size. These objects are stored in the respective caches for 3-KB and 7-KB objects.

The slab-allocation algorithm uses caches to store kernel objects. When a cache is created, a number of objects—which are initially marked as **free**—are allocated to the cache. The number of objects in the cache depends on the size of

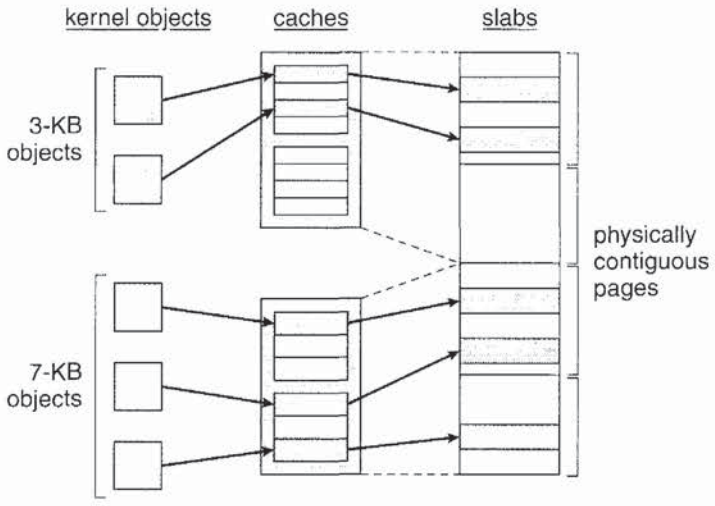


Figure 21.7 Slab allocator in Linux.

the associated slab. For example, a 12-KB slab (comprised of three contiguous 4-KB pages) could store six 2-KB objects. Initially, all objects in the cache are marked as free. When a new object for a kernel data structure is needed, the allocator can assign any free object from the cache to satisfy the request. The object assigned from the cache is marked as **used**.

Let's consider a scenario in which the kernel requests memory from the slab allocator for an object representing a process descriptor. In Linux systems, a process descriptor is of the type `struct task_struct`, which requires approximately 1.7 KB of memory. When the Linux kernel creates a new task, it requests the necessary memory for the `struct task_struct` object from its cache. The cache will fulfill the request using a `struct task_struct` object that has already been allocated in a slab and is marked as free.

In Linux, a slab may be in one of three possible states:

1. **Full**. All objects in the slab are marked as used.
2. **Empty**. All objects in the slab are marked as free.
3. **Partial**. The slab consists of both used and free objects.

The slab allocator first attempts to satisfy the request with a free object in a partial slab. If none exist, a free object is assigned from an empty slab. If no empty slabs are available, a new slab is allocated from contiguous physical pages and assigned to a cache; memory for the object is allocated from this slab.

The other two main subsystems in Linux that do their own management of physical pages are closely related to one another. These are the page cache and the virtual memory system. The **page cache** is the kernel's main cache for block-oriented devices and memory-mapped files and is the main mechanism through which I/O to these devices is performed. Both the native Linux disk-based file systems and the NFS networked file system use the page cache. The page cache caches entire pages of file contents and is not limited to block devices; it can also cache networked data. The virtual memory system manages the contents of each process's virtual address space. These two systems interact closely with one another because reading a page of data into the page cache requires mapping pages in the page cache using the virtual memory system. In the following sections, we look at the virtual memory system in greater detail.

21.6.2 Virtual Memory

The Linux virtual memory system is responsible for maintaining the address space visible to each process. It creates pages of virtual memory on demand and manages the loading of those pages from disk or their swapping back out to disk as required. Under Linux, the virtual memory manager maintains two separate views of a process's address space: as a set of separate regions and as a set of pages.

The first view of an address space is the logical view, describing instructions that the virtual memory system has received concerning the layout of the address space. In this view, the address space consists of a set of nonoverlapping regions, each region representing a continuous, page-aligned subset of the address space. Each region is described internally by a single `vm_area_struct`

structure that defines the properties of the region, including the process's read, write, and execute permissions in the region, and information about any files associated with the region. The regions for each address space are linked into a balanced binary tree to allow fast lookup of the region corresponding to any virtual address.

The kernel also maintains a second, physical view of each address space. This view is stored in the hardware page tables for the process. The page-table entries determine the exact current location of each page of virtual memory, whether it is on disk or in physical memory. The physical view is managed by a set of routines invoked from the kernel's software-interrupt handlers whenever a process tries to access a page that is not currently present in the page tables. Each `vm_area_struct` in the address-space description contains a field that points to a table of functions that implement the key page-management functions for any given virtual memory region. All requests to read or write an unavailable page are eventually dispatched to the appropriate handler in the function table for the `vm_area_struct`, so that the central memory-management routines do not have to know the details of managing each possible type of memory region.

21.6.2.1 Virtual Memory Regions

Linux implements several types of virtual memory regions. The first property that characterizes a type of virtual memory is the backing store for the region, which describes where the pages for a region come from. Most memory regions are backed either by a file or by nothing. A region backed by nothing is the simplest type of virtual memory. Such a region represents **demand-zero memory**: When a process tries to read a page in such a region, it is simply given back a page of memory filled with zeros.

A region backed by a file acts as a viewport onto a section of that file: Whenever the process tries to access a page within that region, the page table is filled with the address of a page within the kernel's page cache corresponding to the appropriate offset in the file. The same page of physical memory is used both by the page cache and by the process's page tables, so any changes made to the file by the file system are immediately visible to any processes that have mapped that file into their address space. Any number of processes can map the same region of the same file, and they will all end up using the same page of physical memory for the purpose.

A virtual memory region is also defined by its reaction to writes. The mapping of a region into the process's address space can be either *private* or *shared*. If a process writes to a privately mapped region, then the pager detects that a copy-on-write is necessary to keep the changes local to the process. In contrast, writes to a shared region result in updating of the object mapped into that region, so that the change will be visible immediately to any other process that is mapping that object.

21.6.2.2 Lifetime of a Virtual Address Space

The kernel will create a new virtual address space in two situations: when a process runs a new program with the `exec()` system call and on creation of a new process by the `fork()` system call. The first case is easy: When a new program is executed, the process is given a new, completely empty virtual

address space. It is up to the routines for loading the program to populate the address space with virtual memory regions.

The second case, creating a new process with `fork()`, involves creating a complete copy of the existing process's virtual address space. The kernel copies the parent process's `vm_area_struct` descriptors, then creates a new set of page tables for the child. The parent's page tables are copied directly into the child's, and the reference count of each page covered is incremented; thus, after the fork, the parent and child share the same physical pages of memory in their address spaces.

A special case occurs when the copying operation reaches a virtual memory region that is mapped privately. Any pages to which the parent process has written within such a region are private, and subsequent changes to these pages by either the parent or the child must not update the page in the other process's address space. When the page-table entries for such regions are copied, they are set to be read only and are marked for copy-on-write. As long as neither process modifies these pages, the two processes share the same page of physical memory. However, if either process tries to modify a copy-on-write page, the reference count on the page is checked. If the page is still shared, then the process copies the page's contents to a brand-new page of physical memory and uses its copy instead. This mechanism ensures that private data pages are shared between processes whenever possible; copies are made only when absolutely necessary.

21.6.2.3 Swapping and Paging

An important task for a virtual memory system is to relocate pages of memory from physical memory out to disk when that memory is needed. Early UNIX systems performed this relocation by swapping out the contents of entire processes at once, but modern versions of UNIX rely more on paging—the movement of individual pages of virtual memory between physical memory and disk. Linux does not implement whole-process swapping; it uses the newer paging mechanism exclusively.

The paging system can be divided into two sections. First, the **policy algorithm** decides which pages to write out to disk and when to write them. Second, the **paging mechanism** carries out the transfer and pages data back into physical memory when they are needed again.

Linux's **pageout policy** uses a modified version of the standard clock (or second-chance) algorithm described in Section 9.4.5.2. Under Linux, a multiple-pass clock is used, and every page has an *age* that is adjusted on each pass of the clock. The age is more precisely a measure of the page's youthfulness, or how much activity the page has seen recently. Frequently accessed pages will attain a higher age value, but the age of infrequently accessed pages will drop toward zero with each pass. This age valuing allows the pager to select pages to page out based on a least frequently used (LFU) policy.

The paging mechanism supports paging both to dedicated swap devices and partitions and to normal files, although swapping to a file is significantly slower due to the extra overhead incurred by the file system. Blocks are allocated from the swap devices according to a bitmap of used blocks, which is maintained in physical memory at all times. The allocator uses a next-fit algorithm to try to write out pages to continuous runs of disk blocks for

improved performance. The allocator records the fact that a page has been paged out to disk by using a feature of the page tables on modern processors: The page-table entry's page-not-present bit is set, allowing the rest of the page-table entry to be filled with an index identifying where the page has been written.

21.6.2.4 Kernel Virtual Memory

Linux reserves for its own internal use a constant, architecture-dependent region of the virtual address space of every process. The page-table entries that map to these kernel pages are marked as protected, so that the pages are not visible or modifiable when the processor is running in user mode. This kernel virtual memory area contains two regions. The first is a static area that contains page-table references to every available physical page of memory in the system, so that a simple translation from physical to virtual addresses occurs when kernel code is run. The core of the kernel, along with all pages allocated by the normal page allocator, resides in this region.

The remainder of the kernel's reserved section of address space is not reserved for any specific purpose. Page-table entries in this address range can be modified by the kernel to point to any other areas of memory. The kernel provides a pair of facilities that allow processes to use this virtual memory. The `vmalloc()` function allocates an arbitrary number of physical pages of memory that may not be physically contiguous into a single region of virtually contiguous kernel memory. The `vremap()` function maps a sequence of virtual addresses to point to an area of memory used by a device driver for memory-mapped I/O.

21.6.3 Execution and Loading of User Programs

The Linux kernel's execution of user programs is triggered by a call to the `exec()` system call. This call commands the kernel to run a new program within the current process, completely overwriting the current execution context with the initial context of the new program. The first job of this system service is to verify that the calling process has permission rights to the file being executed. Once that matter has been checked, the kernel invokes a loader routine to start running the program. The loader does not necessarily load the contents of the program file into physical memory, but it does at least set up the mapping of the program into virtual memory.

There is no single routine in Linux for loading a new program. Instead, Linux maintains a table of possible loader functions, and it gives each such function the opportunity to try loading the given file when an `exec()` system call is made. The initial reason for this loader table was that, between the releases of the 1.0 and 1.2 kernels, the standard format for Linux's binary files was changed. Older Linux kernels understood the `a.out` format for binary files—a relatively simple format common on older UNIX systems. Newer Linux systems use the more modern **ELF** format, now supported by most current UNIX implementations. ELF has a number of advantages over `a.out`, including flexibility and extensibility: New sections can be added to an ELF binary (for example, to add extra debugging information) without causing

the loader routines to become confused. By allowing registration of multiple loader routines, Linux can easily support the ELF and `a.out` binary formats in a single running system.

In Sections 21.6.3.1 and 21.6.3.2, we concentrate exclusively on the loading and running of ELF-format binaries. The procedure for loading `a.out` binaries is simpler but is similar in operation.

21.6.3.1 Mapping of Programs into Memory

Under Linux, the binary loader does not load a binary file into physical memory. Rather, the pages of the binary file are mapped into regions of virtual memory. Only when the program tries to access a given page will a page fault result in the loading of that page into physical memory using demand paging.

It is the responsibility of the kernel's binary loader to set up the initial memory mapping. An ELF-format binary file consists of a header followed by several page-aligned sections. The ELF loader works by reading the header and mapping the sections of the file into separate regions of virtual memory.

Figure 21.8 shows the typical layout of memory regions set up by the ELF loader. In a reserved region at one end of the address space sits the kernel, in its own privileged region of virtual memory inaccessible to normal user-mode programs. The rest of virtual memory is available to applications, which can use the kernel's memory-mapping functions to create regions that map a portion of a file or that are available for application data.

The loader's job is to set up the initial memory mapping to allow the execution of the program to start. The regions that need to be initialized include the stack and the program's text and data regions.

The stack is created at the top of the user-mode virtual memory; it grows downward toward lower-numbered addresses. It includes copies of the

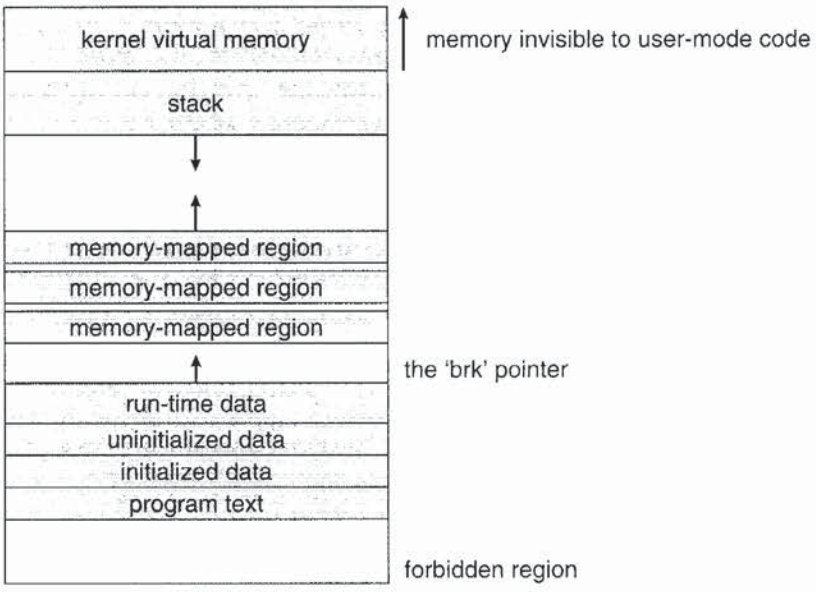


Figure 21.8 Memory layout for ELF programs.

arguments and environment variables given to the program in the `exec()` system call. The other regions are created near the bottom end of virtual memory. The sections of the binary file that contain program text or read-only data are mapped into memory as a write-protected region. Writable initialized data are mapped next; then any uninitialized data are mapped in as a private demand-zero region.

Directly beyond these fixed-sized regions is a variable-sized region that programs can expand as needed to hold data allocated at run time. Each process has a pointer, `brk`, that points to the current extent of this data region, and processes can extend or contract their `brk` region with a single system call —`sbrk()`.

Once these mappings have been set up, the loader initializes the process's program-counter register with the starting point recorded in the ELF header, and the process can be scheduled.

21.6.3.2 Static and Dynamic Linking

Once the program has been loaded and has started running, all the necessary contents of the binary file have been loaded into the process's virtual address space. However, most programs also need to run functions from the system libraries, and these library functions also need to be loaded. In the simplest case, the necessary library functions are embedded directly in the program's executable binary file. Such a program is statically linked to its libraries, and statically linked executables can commence running as soon as they are loaded.

The main disadvantage of static linking is that every program generated must contain copies of exactly the same common system library functions. It is much more efficient, in terms of both physical memory and disk-space usage, to load the system libraries into memory only once. Dynamic linking allows this single loading to happen.

Linux implements dynamic linking in user mode through a special linker library. Every dynamically linked program contains a small, statically linked function that is called when the program starts. This static function just maps the link library into memory and runs the code that the function contains. The link library determines the dynamic libraries required by the program and the names of the variables and functions needed from those libraries by reading the information contained in sections of the ELF binary. It then maps the libraries into the middle of virtual memory and resolves the references to the symbols contained in those libraries. It does not matter exactly where in memory these shared libraries are mapped: They are compiled into **position-independent code (PIC)**, which can run at any address in memory.

21.7 File Systems

Linux retains UNIX's standard file-system model. In UNIX, a file does not have to be an object stored on disk or fetched over a network from a remote file server. Rather, UNIX files can be anything capable of handling the input or output of a stream of data. Device drivers can appear as files, and interprocess-communication channels or network connections also look like files to the user.

The Linux kernel handles all these types of file by hiding the implementation details of any single file type behind a layer of software, the virtual file system (VFS). Here, we first cover the virtual file system and then discuss the standard Linux file system—ext2fs.

21.7.1 The Virtual File System

The Linux VFS is designed around object-oriented principles. It has two components: a set of definitions that specify what file-system objects are allowed to look like and a layer of software to manipulate the objects. The VFS defines four main object types:

- An **inode object** represents an individual file.
- A **file object** represents an open file.
- A **superblock object** represents an entire file system.
- A **dentry object** represents an individual directory entry.

For each of these four object types, the VFS defines a set of operations. Every object of one of these types contains a pointer to a function table. The function table lists the addresses of the actual functions that implement the defined operations for that object. For example, an abbreviated API for some of the file object's operations includes:

- `int open(. . .)` — Open a file.
- `ssize_t read(. . .)` — Read from a file.
- `ssize_t write(. . .)` — Write to a file.
- `int mmap(. . .)` — Memory-map a file.

The complete definition of the file object is specified in the `struct file_operations`, which is located in the file `/usr/include/linux/fs.h`. An implementation of the file object (for a specific file type) is required to implement each function specified in the definition of the file object.

The VFS software layer can perform an operation on one of the file-system objects by calling the appropriate function from the object's function table, without having to know in advance exactly what kind of object it is dealing with. The VFS does not know, or care, whether an inode represents a networked file, a disk file, a network socket, or a directory file. The appropriate function for that file's `read()` operation will always be at the same place in its function table, and the VFS software layer will call that function without caring how the data are actually read.

The inode and file objects are the mechanisms used to access files. An inode object is a data structure containing pointers to the disk blocks that contain the actual file contents, and a file object represents a point of access to the data in an open file. A process cannot access an inode's contents without first obtaining a file object pointing to the inode. The file object keeps track of where in the file the process is currently reading or writing, to keep track of sequential file I/O. It also remembers whether the process asked for write permissions when the file

was opened and tracks the process's activity if necessary to perform adaptive read-ahead, fetching file data into memory before the process requests the data, to improve performance.

File objects typically belong to a single process, but inode objects do not. Even when a file is no longer being used by any processes, its inode object may still be cached by the VFS to improve performance if the file is used again in the near future. All cached file data are linked onto a list in the file's inode object. The inode also maintains standard information about each file, such as the owner, size, and time most recently modified.

Directory files are dealt with slightly differently from other files. The UNIX programming interface defines a number of operations on directories, such as creating, deleting, and renaming a file in a directory. The system calls for these directory operations do not require that the user open the files concerned, unlike the case for reading or writing data. The VFS therefore defines these directory operations in the inode object, rather than in the file object.

The superblock object represents a connected set of files that form a self-contained file system. The operating-system kernel maintains a single superblock object for each disk device mounted as a file system and for each networked file system currently connected. The main responsibility of the superblock object is to provide access to inodes. The VFS identifies every inode by a unique (file-system/inode number) pair, and it finds the inode corresponding to a particular inode number by asking the superblock object to return the inode with that number.

Finally, a dentry object represents a directory entry that may include the name of a directory in the path name of a file (such as `/usr`) or the actual file (such as `stdio.h`). For example, the file `/usr/include/stdio.h` contains the directory entries (1) `/`, (2) `usr`, (3) `include`, and (4) `stdio.h`. Each one of these values is represented by a separate dentry object.

As an example of how dentry objects are used, consider the situation in which a process wishes to open the file with the pathname `/usr/include/stdio.h` using an editor. Because Linux treats directory names as files, translating this path requires first obtaining the inode for the root—`/`. The operating system must then read through this file to obtain the inode for the file `include`. It must continue this process until it obtains the inode for the file `stdio.h`. Because path-name translation can be a time-consuming task, Linux maintains a cache of dentry objects, which is consulted during path-name translation. Obtaining the inode from the dentry cache is considerably faster than having to read the on-disk file.

21.7.2 The Linux ext2fs File System

The standard on-disk file system used by Linux is called **ext2fs**, for historical reasons. Linux was originally programmed with a Minix-compatible file system, to ease exchanging data with the Minix development system, but that file system was severely restricted by 14-character file-name limits and a maximum file-system size of 64 MB. The Minix file system was superseded by a new file system, which was christened the **extended file system (extfs)**. A later redesign of this file system to improve performance and scalability and to add a few missing features led to the **second extended file system (ext2fs)**.

Linuxs ext2fs has much in common with the BSD Fast File System (FFS) (Section A.7.7). It uses a similar mechanism for locating the data blocks belonging to a specific file, storing data-block pointers in indirect blocks throughout the file system with up to three levels of indirection. As in FFS, directory files are stored on disk just like normal files, although their contents are interpreted differently. Each block in a directory file consists of a linked list of entries; each entry contains the length of the entry, the name of a file, and the inode number of the inode to which that entry refers.

The main differences between ext2fs and FFS lie in their disk-allocation policies. In FFS, the disk is allocated to files in blocks of 8 KB. These blocks are subdivided into fragments of 1 KB for storage of small files or partially filled blocks at the ends of files. In contrast, ext2fs does not use fragments at all but performs all its allocations in smaller units. The default block size on ext2fs is 1 KB, although 2-KB and 4-KB blocks are also supported.

To maintain high performance, the operating system must try to perform I/O operations in large chunks whenever possible by clustering physically adjacent I/O requests. Clustering reduces the per-request overhead incurred by device drivers, disks, and disk-controller hardware. A 1-KB I/O request size is too small to maintain good performance, so ext2fs uses allocation policies designed to place logically adjacent blocks of a file into physically adjacent blocks on disk, so that it can submit an I/O request for several disk blocks as a single operation.

The ext2fs allocation policy comes in two parts. As in FFS, an ext2fs file system is partitioned into multiple **block groups**. FFS uses the similar concept of **cylinder groups**, where each group corresponds to a single cylinder of a physical disk. However, modern disk-drive technology packs sectors onto the disk at different densities, and thus with different cylinder sizes, depending on how far the disk head is from the center of the disk. Therefore, fixed-sized cylinder groups do not necessarily correspond to the disk's geometry.

When allocating a file, ext2fs must first select the block group for that file. For data blocks, it attempts to allocate the file to the block group to which the file's inode has been allocated. For inode allocations, it selects the block group in which the file's parent directory resides, for nondirectory files. Directory files are not kept together but rather are dispersed throughout the available block groups. These policies are designed not only to keep related information within the same block group but also to spread out the disk load among the disk's block groups to reduce the fragmentation of any one area of the disk.

Within a block group, ext2fs tries to keep allocations physically contiguous if possible, reducing fragmentation if it can. It maintains a bitmap of all free blocks in a block group. When allocating the first blocks for a new file, it starts searching for a free block from the beginning of the block group; when extending a file, it continues the search from the block most recently allocated to the file. The search is performed in two stages. First, ext2fs searches for an entire free byte in the bitmap; if it fails to find one, it looks for any free bit. The search for free bytes aims to allocate disk space in chunks of at least eight blocks where possible.

Once a free block has been identified, the search is extended backward until an allocated block is encountered. When a free byte is found in the bitmap, this backward extension prevents ext2fs from leaving a hole between the most recently allocated block in the previous nonzero byte and the zero byte found.

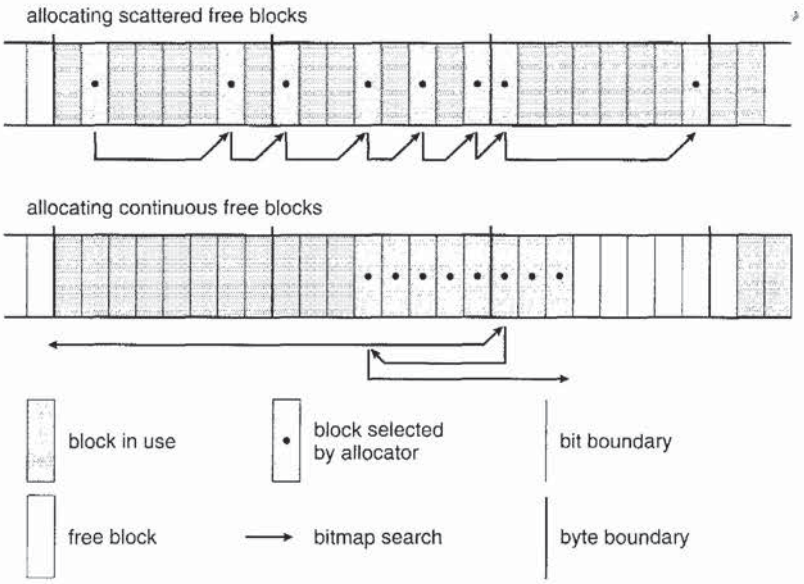


Figure 21.9 ext2fs block-allocation policies.

Once the next block to be allocated has been found by either bit or byte search, ext2fs extends the allocation forward for up to eight blocks and **preallocates** these extra blocks to the file. This preallocation helps to reduce fragmentation during interleaved writes to separate files and also reduces the CPU cost of disk allocation by allocating multiple blocks simultaneously. The preallocated blocks are returned to the free-space bitmap when the file is closed.

Figure 21.9 illustrates the allocation policies. Each row represents a sequence of set and unset bits in an allocation bitmap, indicating used and free blocks on disk. In the first case, if we can find any free blocks sufficiently near the start of the search, then we allocate them no matter how fragmented they may be. The fragmentation is partially compensated for by the fact that the blocks are close together and can probably all be read without any disk seeks, and allocating them all to one file is better in the long run than allocating isolated blocks to separate files once large free areas become scarce on disk. In the second case, we have not immediately found a free block close by, so we search forward for an entire free byte in the bitmap. If we allocated that byte as a whole, we would end up creating a fragmented area of free space before it, so before allocating we back up to make this allocation flush with the allocation preceding it, and then we allocate forward to satisfy the default allocation of eight blocks.

21.7.3 Journaling

Many different types of file systems are available for Linux systems. One popular feature in a file system is **journaling**, whereby modifications to the file system are sequentially written to a journal. A set of operations that performs a specific task is a **transaction**. Once a transaction is written to the journal, it is considered to be committed, and the system call modifying the file system

(i.e. `write()`) can return to the user process, allowing it to continue execution. Meanwhile, the journal entries relating to the transaction are replayed across the actual file-system structures. As the changes are made, a pointer is updated to indicate which actions have completed and which are still incomplete. When an entire committed transaction is completed, it is removed from the journal. The journal, which is actually a circular buffer, may be in a separate section of the file system, or it may even be on a separate disk spindle. It is more efficient, but more complex, to have it under separate read–write heads, thereby decreasing head contention and seek times.

If the system crashes, there will be zero or more transactions in the journal. Those transactions were never completed to the file system even though they were committed by the operating system, so they must be completed. The transactions can be executed from the pointer until the work is complete, and the file-system structures remain consistent. The only problem occurs when a transaction has been aborted. That is, it was not committed before the system crashed. Any changes from those transactions that were applied to the file system must be undone, again preserving the consistency of the file system. This recovery is all that is needed after a crash, eliminating all problems with consistency checking.

Journaling file systems are also typically faster than non-journaling systems, as updates proceed much faster when they are applied to the in-memory journal rather than directly to the on-disk data structures. The reason for this improvement is found in the performance advantage of sequential I/O over random I/O. The costly synchronous random writes to the file system are turned into much less costly synchronous sequential writes to the file system's journal. Those changes in turn are replayed asynchronously via random writes to the appropriate structures. The overall result is a significant gain in performance of file system metadata-oriented operations, such as file creation and deletion.

Journaling is not provided in ext2fs. It is provided, however, in another common file system available for Linux systems, **ext3**, which is based on ext2fs.

21.7.4 The Linux proc File System

The flexibility of the Linux VFS enables us to implement a file system that does not store data persistently at all but rather simply provides an interface to some other functionality. The Linux **process file system**, known as the `/proc` file system, is an example of a file system whose contents are not actually stored anywhere but are computed on demand according to user file I/O requests.

A `/proc` file system is not unique to Linux. SVR4 UNIX introduced a `/proc` file system as an efficient interface to the kernel's process debugging support: Each subdirectory of the file system corresponded not to a directory on any disk but rather to an active process on the current system. A listing of the file system reveals one directory per process, with the directory name being the ASCII decimal representation of the process's unique process identifier (PID).

Linux implements such a `/proc` file system but extends it greatly by adding a number of extra directories and text files under the file system's root directory. These new entries correspond to various statistics about the kernel and the associated loaded drivers. The `/proc` file system provides a way for programs to access this information as plain text files, which the standard

UNIX user environment provides powerful tools to process. For example, in the past, the traditional UNIX `ps` command for listing the states of all running processes has been implemented as a privileged process that reads the process state directly from the kernel's virtual memory. Under Linux, this command is implemented as an entirely unprivileged program that simply parses and formats the information from `/proc`.

The `/proc` file system must implement two things: a directory structure and the file contents within. Given that a UNIX file system is defined as a set of file and directory inodes identified by their inode numbers, the `/proc` file system must define a unique and persistent inode number for each directory and the associated files. Once such a mapping exists, it can use this inode number to identify just what operation is required when a user tries to read from a particular file inode or to perform a lookup in a particular directory inode. When data are read from one of these files, the `/proc` file system will collect the appropriate information, format it into textual form, and place it into the requesting process's read buffer.

The mapping from inode number to information type splits the inode number into two fields. In Linux, a PID is 16 bits wide, but an inode number is 32 bits. The top 16 bits of the inode number are interpreted as a PID, and the remaining bits define what type of information is being requested about that process.

A PID of zero is not valid, so a zero PID field in the inode number is taken to mean that this inode contains global—rather than process-specific—information. Separate global files exist in `/proc` to report information such as the kernel version, free memory, performance statistics, and drivers currently running.

Not all the inode numbers in this range are reserved. The kernel can allocate new `/proc` inode mappings dynamically, maintaining a bitmap of allocated inode numbers. It also maintains a tree data structure of registered global `/proc` file-system entries. Each entry contains the file's inode number, file name, and access permissions, along with the special functions used to generate the file's contents. Drivers can register and deregister entries in this tree at any time, and a special section of the tree—appearing under the */proc/sys* directory—is reserved for kernel variables. Files under this tree are dealt with by a set of common handlers that allow both reading and writing of these variables, so a system administrator can tune the value of kernel parameters simply by writing the new desired values out in ASCII decimal to the appropriate file.

To allow efficient access to these variables from within applications, the */proc/sys* subtree is made available through a special system call, `sysctl()`, that reads and writes the same variables in binary, rather than in text, without the overhead of the file system. `sysctl()` is not an extra facility; it simply reads the `/proc` dynamic entry tree to decide to which variables the application is referring.

21.8 Input and Output

To the user, the I/O system in Linux looks much like that in any UNIX system. That is, to the extent possible, all device drivers appear as normal files. A user can open an access channel to a device in the same way she opens any

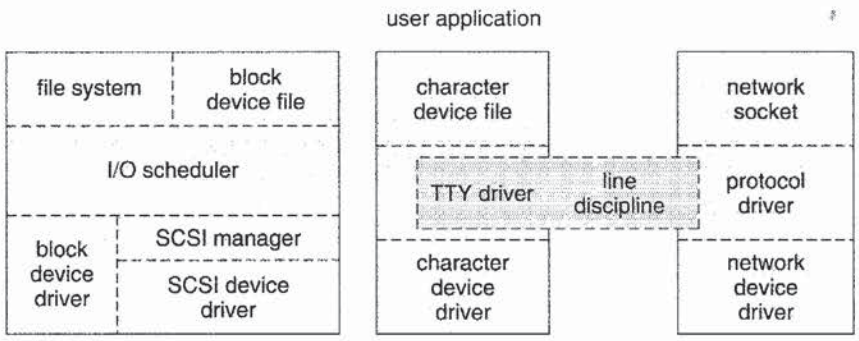


Figure 21.10 Device-driver block structure.

other file—devices can appear as objects within the file system. The system administrator can create special files within a file system that contain references to a specific device driver, and a user opening such a file will be able to read from and write to the device referenced. By using the normal file-protection system, which determines who can access which file, the administrator can set access permissions for each device.

Linux splits all devices into three classes: block devices, character devices, and network devices. Figure 21.10 illustrates the overall structure of the device-driver system.

Block devices include all devices that allow random access to completely independent, fixed-sized blocks of data, including hard disks and floppy disks, CD-ROMs, and flash memory. Block devices are typically used to store file systems, but direct access to a block device is also allowed so that programs can create and repair the file system that the device contains. Applications can also access these block devices directly if they wish; for example, a database application may prefer to perform its own, fine-tuned laying out of data onto the disk, rather than using the general-purpose file system.

Character devices include most other devices, such as mice and keyboards. The fundamental difference between block and character devices is random access—block devices may be accessed randomly, while character devices are only accessed serially. For example, seeking to a certain position in a file might be supported for a DVD but makes no sense to a pointing device such as a mouse.

Network devices are dealt with differently from block and character devices. Users cannot directly transfer data to network devices; instead, they must communicate indirectly by opening a connection to the kernel's networking subsystem. We discuss the interface to network devices separately in Section 21.10.

21.8.1 Block Devices

Block devices provide the main interface to all disk devices in a system. Performance is particularly important for disks, and the block-device system must provide functionality to ensure that disk access is as fast as possible. This functionality is achieved through the scheduling of I/O operations.

In the context of block devices, a **block** represents the unit with which the kernel performs I/O. When a block is read into memory, it is stored in a **buffer**. The **request manager** is the layer of software that manages the reading and writing of buffer contents to and from a block-device driver.

A separate list of requests is kept for each block-device driver. Traditionally, these requests have been scheduled according to a unidirectional-elevator (C-SCAN) algorithm that exploits the order in which requests are inserted in and removed from the per-device lists. The request lists are maintained in sorted order of increasing starting-sector number. When a request is accepted for processing by a block-device driver, it is not removed from the list. It is removed only after the I/O is complete, at which point the driver continues with the next request in the list, even if new requests have been inserted into the list before the active request. As new I/O requests are made, the request manager attempts to merge requests in the per-device lists.

The scheduling of I/O operations changed somewhat with version 2.6 of the kernel. The fundamental problem with the elevator algorithm is that I/O operations concentrated in a specific region of the disk can result in starvation of requests that need to occur in other regions of the disk. The **deadline I/O scheduler** used in version 2.6 works similarly to the elevator algorithm except that it also associates a deadline with each request, thus addressing the starvation issue. By default, the deadline for read requests is 0.5 second and that for write requests is 5 seconds. The deadline scheduler maintains a **sorted queue** of pending I/O operations sorted by sector number. However, it also maintains two other queues—a **read queue** for read operations and a **write queue** for write operations. These two queues are ordered according to deadline. Every I/O request is placed in both the sorted queue and either the read or the write queue, as appropriate. Ordinarily, I/O operations occur from the sorted queue. However, if a deadline expires for a request in either the read or the write queue, I/O operations are scheduled from the queue containing the expired request. This policy ensures that an I/O operation will wait no longer than its expiration time.

21.8.2 Character Devices

A character-device driver can be almost any device driver that does not offer random access to fixed blocks of data. Any character-device drivers registered to the Linux kernel must also register a set of functions that implement the file I/O operations that the driver can handle. The kernel performs almost no preprocessing of a file read or write request to a character device; it simply passes the request to the device in question and lets the device deal with the request.

The main exception to this rule is the special subset of character-device drivers that implement terminal devices. The kernel maintains a standard interface to these drivers by means of a set of `tty_struct` structures. Each of these structures provides buffering and flow control on the data stream from the terminal device and feeds those data to a line discipline.

A **line discipline** is an interpreter for the information from the terminal device. The most common line discipline is the `tty` discipline, which glues the terminal's data stream onto the standard input and output streams of a user's running processes, allowing those processes to communicate directly with the

user's terminal. This job is complicated by the fact that several such processes may be running simultaneously, and the `tty` line discipline is responsible for attaching and detaching the terminal's input and output from the various processes connected to it as those processes are suspended or awakened by the user.

Other line disciplines also are implemented that have nothing to do with I/O to a user process. The PPP and SLIP networking protocols are ways of encoding a networking connection over a terminal device such as a serial line. These protocols are implemented under Linux as drivers that at one end appear to the terminal system as line disciplines and at the other end appear to the networking system as network-device drivers. After one of these line disciplines has been enabled on a terminal device, any data appearing on that terminal will be routed directly to the appropriate network-device driver.

21.9 Interprocess Communication

UNIX provides a rich environment for processes to communicate with each other. Communication may be just a matter of letting another process know that some event has occurred, or it may involve transferring data from one process to another.

21.9.1 Synchronization and Signals

The standard UNIX mechanism for informing a process that an event has occurred is the **signal**. Signals can be sent from any process to any other process, with restrictions on signals sent to processes owned by another user. However, a limited number of signals are available, and they cannot carry information: Only the fact that a signal occurred is available to a process. Signals are not generated only by processes. The kernel also generates signals internally; for example, it can send a signal to a server process when data arrive on a network channel, to a parent process when a child terminates, or to a waiting process when a timer expires.

Internally, the Linux kernel does not use signals to communicate with processes running in kernel mode. If a kernel-mode process is expecting an event to occur, it will not normally use signals to receive notification of that event. Rather, communication about incoming asynchronous events within the kernel is performed through the use of scheduling states and `wait_queue` structures. These mechanisms allow kernel-mode processes to inform one another about relevant events, and they also allow events to be generated by device drivers or by the networking system. Whenever a process wants to wait for some event to complete, it places itself on a **wait queue** associated with that event and tells the scheduler that it is no longer eligible for execution. Once the event has completed, it will wake up every process on the wait queue. This procedure allows multiple processes to wait for a single event. For example, if several processes are trying to read a file from a disk, then they will all be awakened once the data have been read into memory successfully.

Although signals have always been the main mechanism for communicating asynchronous events among processes, Linux also implements the semaphore mechanism of System V UNIX. A process can wait on a semaphore

as easily as it can wait for a signal, but semaphores have two advantages: Large numbers of semaphores can be shared among multiple independent processes, and operations on multiple semaphores can be performed atomically. Internally, the standard Linux wait queue mechanism synchronizes processes that are communicating with semaphores.

21.9.2 Passing of Data Among Processes

Linux offers several mechanisms for passing data among processes. The standard UNIX **pipe** mechanism allows a child process to inherit a communication channel from its parent; data written to one end of the pipe can be read at the other. Under Linux, pipes appear as just another type of inode to virtual-file-system software, and each pipe has a pair of wait queues to synchronize the reader and writer. UNIX also defines a set of networking facilities that can send streams of data to both local and remote processes. Networking is covered in Section 21.10.

Two other methods of sharing data among processes are available. First, shared memory offers an extremely fast way to communicate large or small amounts of data; any data written by one process to a shared memory region can be read immediately by any other process that has mapped that region into its address space. The main disadvantage of shared memory is that, on its own, it offers no synchronization: A process can neither ask the operating system whether a piece of shared memory has been written to nor suspend execution until such a write occurs. Shared memory becomes particularly powerful when used in conjunction with another interprocess-communication mechanism that provides the missing synchronization.

A shared-memory region in Linux is a persistent object that can be created or deleted by processes. Such an object is treated as though it were a small independent address space. The Linux paging algorithms can elect to page out to disk shared-memory pages, just as they can page out a process's data pages. The shared-memory object acts as a backing store for shared-memory regions, just as a file can act as a backing store for a memory-mapped memory region. When a file is mapped into a virtual-address-space region, then any page faults that occur cause the appropriate page of the file to be mapped into virtual memory. Similarly, shared-memory mappings direct page faults to map in pages from a persistent shared-memory object. Also just as for files, shared-memory objects remember their contents even if no processes are currently mapping them into virtual memory.

21.10 Network Structure

Networking is a key area of functionality for Linux. Not only does Linux support the standard Internet protocols used for most UNIX-to-UNIX communications, but it also implements a number of protocols native to other, non-UNIX operating systems. In particular, since Linux was originally implemented primarily on PCs, rather than on large workstations or on server-class systems, it supports many of the protocols typically used on PC networks, such as AppleTalk and IPX.

Internally, networking in the Linux kernel is implemented by three layers of software:

1. The socket interface
2. Protocol drivers
3. Network-device drivers

User applications perform all networking requests through the socket interface. This interface is designed to look like the 4.3 BSD socket layer, so that any programs designed to make use of Berkeley sockets will run on Linux without any source-code changes. This interface is described in Section A.9.1. The BSD socket interface is sufficiently general to represent network addresses for a wide range of networking protocols. This single interface is used in Linux to access not just those protocols implemented on standard BSD systems but all the protocols supported by the system.

The next layer of software is the protocol stack, which is similar in organization to BSD's own framework. Whenever any networking data arrive at this layer, either from an application's socket or from a network-device driver, the data are expected to have been tagged with an identifier specifying which network protocol they contain. Protocols can communicate with one another if they desire; for example, within the Internet protocol set, separate protocols manage routing, error reporting, and reliable retransmission of lost data.

The protocol layer may rewrite packets, create new packets, split or reassemble packets into fragments, or simply discard incoming data. Ultimately, once it has finished processing a set of packets, it passes them on, up to the socket interface if the data are destined for a local connection or downward to a device driver if the packet needs to be transmitted remotely. The protocol layer decides to which socket or device to send the packet.

All communication between the layers of the networking stack is performed by passing single `skbuff` structures. An `skbuff` contains a set of pointers into a single continuous area of memory, representing a buffer inside which network packets can be constructed. The valid data in an `skbuff` do not need to start at the beginning of the `skbuff`'s buffer, and they do not need to run to the end. The networking code can add data to or trim data from either end of the packet, as long as the result still fits into the `skbuff`. This capacity is especially important on modern microprocessors, where improvements in CPU speed have far outstripped the performance of main memory. The `skbuff` architecture allows flexibility in manipulating packet headers and checksums while avoiding any unnecessary data copying.

The most important set of protocols in the Linux networking system is the TCP/IP protocol suite. This suite comprises a number of separate protocols. The IP protocol implements routing between different hosts anywhere on the network. On top of the routing protocol are built the UDP, TCP, and ICMP protocols. The UDP protocol carries arbitrary individual datagrams between hosts. The TCP protocol implements reliable connections between hosts with guaranteed in-order delivery of packets and automatic retransmission of lost data. The ICMP protocol is used to carry various error and status messages between hosts.

Packets (`skbuffs`) arriving at the networking stack's protocol software are expected to be already tagged with an internal identifier indicating to which protocol the packet is relevant. Different networking-device drivers encode the protocol type in different ways over their communications media; thus, the protocol for incoming data must be identified in the device driver. The device driver uses a hash table of known networking-protocol identifiers to look up the appropriate protocol and passes the packet to that protocol. New protocols can be added to the hash table as kernel-loadable modules.

Incoming IP packets are delivered to the IP driver. The job of this layer is to perform routing. After deciding where the packet is destined, it forwards the packet to the appropriate internal protocol driver to be delivered locally or injects it back into a selected network-device-driver queue to be forwarded to another host. It performs the routing decision using two tables: the persistent forwarding information base (FIB) and a cache of recent routing decisions. The FIB holds routing-configuration information and can specify routes based either on a specific destination address or on a wildcard representing multiple destinations. The FIB is organized as a set of hash tables indexed by destination address; the tables representing the most specific routes are always searched first. Successful lookups from this table are added to the route-caching table, which caches routes only by specific destination; no wildcards are stored in the cache, so lookups can be made quickly. An entry in the route cache expires after a fixed period with no hits.

At various stages, the IP software passes packets to a separate section of code for **firewall management**—selective filtering of packets according to arbitrary criteria, usually for security purposes. The firewall manager maintains a number of separate **firewall chains** and allows an `skbuff` to be matched against any chain. Chains are reserved for separate purposes: One is used for forwarded packets, one for packets being input to this host, and one for data generated at this host. Each chain is held as an ordered list of rules, where a rule specifies one of a number of possible firewall-decision functions plus some arbitrary data to match against.

Two other functions performed by the IP driver are disassembly and reassembly of large packets. If an outgoing packet is too large to be queued to a device, it is simply split up into smaller **fragments**, which are all queued to the driver. At the receiving host, these fragments must be reassembled. The IP driver maintains an `ipfrag` object for each fragment awaiting reassembly and an `ipq` for each datagram being assembled. Incoming fragments are matched against each known `ipq`. If a match is found, the fragment is added to it; otherwise, a new `ipq` is created. Once the final fragment has arrived for a `ipq`, a completely new `skbuff` is constructed to hold the new packet, and this packet is passed back into the IP driver.

Packets identified by the IP as destined for this host are passed on to one of the other protocol drivers. The UDP and TCP protocols share a means of associating packets with source and destination sockets: Each connected pair of sockets is uniquely identified by its source and destination addresses and by the source and destination port numbers. The socket lists are linked onto hash tables keyed on these four address–port values for socket lookup on incoming packets. The TCP protocol has to deal with unreliable connections, so it maintains ordered lists of unacknowledged outgoing packets to retransmit

after a timeout and of incoming out-of-order packets to be presented to the socket when the missing data have arrived.

21.11 Security

Linux's security model is closely related to typical UNIX security mechanisms. The security concerns can be classified in two groups:

1. **Authentication**. Making sure that nobody can access the system without first proving that she has entry rights
2. **Access control**. Providing a mechanism for checking whether a user has the right to access a certain object and preventing access to objects as required

21.11.1 Authentication

Authentication in UNIX has typically been performed through the use of a publicly readable password file. A user's password is combined with a random "salt" value, and the result is encoded with a one-way transformation function and stored in the password file. The use of the one-way function means that the original password cannot be deduced from the password file except by trial and error. When a user presents a password to the system, the password is recombined with the salt value stored in the password file and passed through the same one-way transformation. If the result matches the contents of the password file, then the password is accepted.

Historically, UNIX implementations of this mechanism have had several problems. Passwords were often limited to eight characters, and the number of possible salt values was so low that an attacker could easily combine a dictionary of commonly used passwords with every possible salt value and have a good chance of matching one or more passwords in the password file, gaining unauthorized access to any accounts compromised as a result. Extensions to the password mechanism have been introduced that keep the encrypted password secret in a file that is not publicly readable, that allow longer passwords, or that use more secure methods of encoding the password. Other authentication mechanisms have been introduced that limit the times during which a user is permitted to connect to the system or to distribute authentication information to all the related systems in a network.

A new security mechanism has been developed by UNIX vendors to address authentication problems. The **pluggable authentication modules (PAM)** system is based on a shared library that can be used by any system component that needs to authenticate users. An implementation of this system is available under Linux. PAM allows authentication modules to be loaded on demand as specified in a system-wide configuration file. If a new authentication mechanism is added at a later date, it can be added to the configuration file, and all system components will immediately be able to take advantage of it. PAM modules can specify authentication methods, account restrictions, session-setup functions, and password-changing functions (so that, when users change their passwords, all the necessary authentication mechanisms can be updated at once).

21.11.2 Access Control

Access control under UNIX systems, including Linux, is performed through the use of unique numeric identifiers. A user identifier (uid) identifies a single user or a single set of access rights. A group identifier (gid) is an extra identifier that can be used to identify rights belonging to more than one user.

Access control is applied to various objects in the system. Every file available in the system is protected by the standard access-control mechanism. In addition, other shared objects, such as shared-memory sections and semaphores, employ the same access system.

Every object in a UNIX system under user and group access control has a single uid and a single gid associated with it. User processes also have a single uid, but they may have more than one gid. If a process's uid matches the uid of an object, then the process has **user rights** or **owner rights** to that object. If the uids do not match but any of the process's gids match the object's gid, then **group rights** are conferred; otherwise, the process has **world rights** to the object.

Linux performs access control by assigning objects a **protection mask** that specifies which access modes—read, write, or execute—are to be granted to processes with owner, group, or world access. Thus, the owner of an object might have full read, write, and execute access to a file; other users in a certain group might be given read access but denied write access; and everybody else might be given no access at all.

The only exception is the privileged **root** uid. A process with this special uid is granted automatic access to any object in the system, bypassing normal access checks. Such processes are also granted permission to perform privileged operations, such as reading any physical memory or opening reserved network sockets. This mechanism allows the kernel to prevent normal users from accessing these resources: Most of the kernel's key internal resources are implicitly owned by the root uid.

Linux implements the standard UNIX `setuid` mechanism described in Section A.3.2. This mechanism allows a program to run with privileges different from those of the user running the program. For example, the `lpr` program (which submits a job onto a print queue) has access to the system's print queues even if the user running that program does not. The UNIX implementation of `setuid` distinguishes between a process's *real* and *effective* uid: The real uid is that of the user running the program; the effective uid is that of the file's owner.

Under Linux, this mechanism is augmented in two ways. First, Linux implements the POSIX specification's `saved user-id` mechanism, which allows a process to drop and reacquire its effective uid repeatedly. For security reasons, a program may want to perform most of its operations in a safe mode, waiving the privileges granted by its `setuid` status, but may wish to perform selected operations with all its privileges. Standard UNIX implementations achieve this capacity only by swapping the real and effective uids; the previous effective uid is remembered, but the program's real uid does not always correspond to the uid of the user running the program. Saved uids allow a process to set its effective uid to its real uid and then back to the previous value of its effective uid without having to modify the real uid at any time.

The second enhancement provided by Linux is the addition of a process characteristic that grants just a subset of the rights of the effective uid. The

fsuid and **fsgid** process properties are used when access rights are granted to files. The appropriate property is set every time the effective uid or gid is set. However, the fsuid and fsgid can be set independently of the effective ids, allowing a process to access files on behalf of another user without taking on the identity of that other user in any other way. Specifically, server processes can use this mechanism to serve files to a certain user without the process becoming vulnerable to being killed or suspended by that user.

Finally, Linux provides a mechanism for flexible passing of rights from one program to another—a mechanism that has become common in modern versions of UNIX. When a local network socket has been set up between any two processes on the system, either of those processes may send to the other process a file descriptor for one of its open files; the other process receives a duplicate file descriptor for the same file. This mechanism allows a client to pass access to a single file selectively to some server process without granting that process any other privileges. For example, it is no longer necessary for a print server to be able to read all the files of a user who submits a new print job; the print client could simply pass the server file descriptors for any files to be printed, denying the server access to any of the user's other files.

21.12 Summary

Linux is a modern, free operating system based on UNIX standards. It has been designed to run efficiently and reliably on common PC hardware; it also runs on a variety of other platforms. It provides a programming interface and user interface compatible with standard UNIX systems and can run a large number of UNIX applications, including an increasing number of commercially supported applications.

Linux has not evolved in a vacuum. A complete Linux system includes many components that were developed independently of Linux. The core Linux operating-system kernel is entirely original, but it allows much existing free UNIX software to run, resulting in an entire UNIX-compatible operating system free from proprietary code.

The Linux kernel is implemented as a traditional monolithic kernel for performance reasons, but it is modular enough in design to allow most drivers to be dynamically loaded and unloaded at run time.

Linux is a multiuser system, providing protection between processes and running multiple processes according to a time-sharing scheduler. Newly created processes can share selective parts of their execution environment with their parent processes, allowing multithreaded programming. Interprocess communication is supported by both System V mechanisms—message queues, semaphores, and shared memory—and BSD's socket interface. Multiple networking protocols can be accessed simultaneously through the socket interface.

To the user, the file system appears as a hierarchical directory tree that obeys UNIX semantics. Internally, Linux uses an abstraction layer to manage multiple different file systems. Device-oriented, networked, and virtual file systems are supported. Device-oriented file systems access disk storage through a page cache that is unified with the virtual memory system.

The memory-management system uses page sharing and copy-on-write to minimize the duplication of data shared by different processes. Pages are loaded on demand when they are first referenced and are paged back out to backing store according to an LFU algorithm if physical memory needs to be reclaimed.

Exercises

21.1 What are the advantages and disadvantages of writing an operating system in a high-level language, such as C?

21.2 In what circumstances is the system-call sequence `fork()` `exec()` most appropriate? When is `vfork()` preferable?

21.3 What socket type should be used to implement an intercomputer file-transfer program? What type should be used for a program that periodically tests to see whether another computer is up on the network? Explain your answer.

21.4 Linux runs on a variety of hardware platforms. What steps must the Linux developers take to ensure that the system is portable to different processors and memory-management architectures, and to minimize the amount of architecture-specific kernel code?

21.5 What are the advantages and disadvantages of making only some of the symbols defined inside a kernel accessible to a loadable kernel module?

21.6 What are the primary goals of the conflict-resolution mechanism used by the Linux kernel for loading kernel modules?

21.7 Discuss how the clone() operation supported by Linux is used to support both processes and threads.

21.8 Would one classify Linux threads as user-level threads or as kernel-level threads? Support your answer with the appropriate arguments.

21.9 What extra costs are incurred by the creation and scheduling of a process, compared with the cost of a cloned thread?

21.10 The Linux scheduler implements *soft* real-time scheduling. What features necessary for certain real-time programming tasks are missing? How might they be added to the kernel?

21.11 Under what circumstances would an user process request an operation that results in the allocation of a demand-zero memory region?

21.12 What scenarios would cause a page of memory to be mapped into an user program's address space with the copy-on-write attribute enabled?

21.13 In Linux, shared libraries perform many operations central to the operating system. What is the advantage of keeping this functionality out of the kernel? Are there any drawbacks? Explain your answer.

21.14 The directory structure of a Linux operating system could comprise of files corresponding to different file systems, including the Linux /proc

file system. What are the implications of having to support different file-system types on the structure of the Linux kernel?

21.15 In what ways does the Linux setuid feature differ from the setuid feature in standard Unix?

21.16 The Linux source code is freely and widely available over the Internet or from CD-ROM vendors. What are three implications of this availability for the security of the Linux system?

Bibliographical Notes

The Linux system is a product of the Internet; as a result, much of the available documentation on Linux is available in some form on the Internet. The following key sites reference most of the useful information available:

- The Linux Cross-Reference Pages at http://lxr.linux.no/ maintain current listings of the Linux kernel, browsable via the Web and fully cross-referenced.
- Linux-HQ at http://www.linuxhq.com/ hosts a large amount of information relating to the Linux 2.x kernels. This site also includes links to the home pages of most Linux distributions, as well as archives of the major mailing lists.
- The Linux Documentation Project at http://sunsite.unc.edu/linux/ lists many books on Linux that are available in source format as part of the Linux Documentation Project. The project also hosts the Linux *How-To* guides, which contain a series of hints and tips relating to aspects of Linux.
- The *Kernel Hackers' Guide* is an Internet-based guide to kernel internals in general. This constantly expanding site is located at http://www.redhat.com:8080/HyperNews/get/khg.html.
- The Kernel Newbies website (http://www.kernelnewbies.org/) provides a resource for introducing the Linux kernel to newcomers.

Many mailing lists devoted to Linux are also available. The most important are maintained by a mailing-list manager that can be reached at the e-mail address `majordomo@vger.rutgers.edu`. Send e-mail to this address with the single line "help" in the mail's body for information on how to access the list server and to subscribe to any lists.

Finally, the Linux system itself can be obtained over the Internet. Complete Linux distributions can be obtained from the home sites of the companies concerned, and the Linux community also maintains archives of current system components at several places on the Internet. The most important are these:

- ftp://tsx-11.mit.edu/pub/linux/
- ftp://sunsite.unc.edu/pub/Linux/
- ftp://linux.kernel.org/pub/linux/

In addition to investigating Internet resources, you can read about the internals of the Linux kernel in Bovet and Cesati [2002] and Love [2004].



Windows XP

The Microsoft Windows XP operating system is a 32/64-bit preemptive multitasking operating system for AMD K6/K7, Intel IA32/IA64, and later microprocessors. The successor to Windows NT and Windows 2000, Windows XP is also intended to replace the Windows 95/98 operating system. Key goals for the system are security, reliability, ease of use, Windows and POSIX application compatibility, high performance, extensibility, portability, and international support. In this chapter, we discuss the key goals of Windows XP, the layered architecture of the system that makes it so easy to use, the file system, the networking features, and the programming interface.

CHAPTER OBJECTIVES

- To explore the principles upon which Windows XP is designed and the specific components involved in the system.
- To understand how Windows XP can run programs designed for other operating systems.
- To provide a detailed explanation of the Windows XP file system.
- To illustrate the networking protocols supported in Windows XP.
- To cover the interface available to system and application programmers.

22.1 History

In the mid-1980s, Microsoft and IBM cooperated to develop the OS/2 operating system, which was written in assembly language for single-processor Intel 80286 systems. In 1988, Microsoft decided to make a fresh start and to develop a "new technology" (or NT) portable operating system that supported both the OS/2 and POSIX application-programming interfaces (APIs). In October 1988, Dave Cutler, the architect of the DEC VAX/VMS operating system, was hired and given the charter of building this new operating system.

Originally, the team planned for NT to use the OS/2 API as its native environment, but during development, NT was changed to use the 32-bit

Windows API (or Win32 API), reflecting the popularity of Windows 3.0. The first versions of NT were Windows NT 3.1 and Windows NT 3.1 Advanced Server. (At that time, 16-bit Windows was at version 3.1.) Windows NT version 4.0 adopted the Windows 95 user interface and incorporated Internet web-server and web-browser software. In addition, user-interface routines and all graphics code were moved into the kernel to improve performance, with the side effect of decreased system reliability. Although previous versions of NT had been ported to other microprocessor architectures, the Windows 2000 version, released in February 2000, discontinued support for other than Intel (and compatible) processors due to marketplace factors. Windows 2000 incorporated significant changes over Windows NT. It added Active Directory (an X.500-based directory service), better networking and laptop support, support for plug-and-play devices, a distributed file system, and support for more processors and more memory.

In October 2001, Windows XP was released as both an update to the Windows 2000 desktop operating system and a replacement for Windows 95/98. In 2002, the server versions of Windows XP became available (called Windows .Net Server). Windows XP updates the graphical user interface (GUI) with a visual design that takes advantage of more recent hardware advances and many new **ease-of-use** features. Numerous features have been added to automatically repair problems in applications and the operating system itself. Windows XP provides better networking and device experience (including zero-configuration wireless, instant messaging, streaming media, and digital photography/video), dramatic performance improvements both for the desktop and large multiprocessors, and better reliability and security than even Windows 2000.

Windows XP uses a client–server architecture (like Mach) to implement multiple operating-system personalities, such as Win32 API and POSIX, with user-level processes called subsystems. The subsystem architecture allows enhancements to be made to one operating-system personality without affecting the application compatibility of any others.

Windows XP is a multiuser operating system, supporting simultaneous access through distributed services or through multiple instances of the graphical user interface via the Windows terminal server. The server versions of Windows XP support simultaneous terminal server sessions from Windows desktop systems. The desktop versions of terminal server multiplex the keyboard, mouse, and monitor between virtual terminal sessions for each logged-on user. This feature, called fast user switching, allows users to preempt each other at the console of a PC without having to log off and onto the system.

Windows XP is the first version of Windows to ship a 64-bit version. The native NT file system (NTFS) and many of the Win32 APIs have always used 64-bit integers where appropriate—so the major extension to 64-bit in Windows XP is support for large addresses.

There are two desktop versions of Windows XP. Windows XP Professional is the premium desktop system for power users at work and at home. For home users migrating from Windows 95/98, Windows XP Personal provides the reliability and ease of use of Windows XP, but lacks the more advanced features needed to work seamlessly with Active Directory or run POSIX applications.

The members of the Windows .Net Server family use the same core components as the desktop versions but add a range of features needed for

uses such as webserver farms, print/file servers, clustered systems, and large datacenter machines. The large datacenter machines can have up to 64 GB of memory and 32 processors on IA32 systems and 128 GB and 64 processors on IA64 systems.

22.2 Design Principles

Microsoft's design goals for Windows XP include security, reliability, Windows and POSIX application compatibility, high performance, extensibility, portability, and international support.

22.2.1 Security

Windows XP **security** goals required more than just adherence to the design standards that enabled Windows NT 4.0 to receive a C-2 security classification from the U.S. government (which signifies a moderate level of protection from defective software and malicious attacks). Extensive code review and testing were combined with sophisticated automatic analysis tools to identify and investigate potential defects that might represent security vulnerabilities.

22.2.2 Reliability

Windows 2000 was the most reliable, stable operating system Microsoft had ever shipped to that point. Much of this reliability came from maturity in the source code, extensive stress testing of the system, and automatic detection of many serious errors in drivers. The **reliability** requirements for Windows XP were even more stringent. Microsoft used extensive manual and automatic code review to identify over 63,000 lines in the source files that might contain issues not detected by testing and then set about reviewing each area to verify that the code was indeed correct.

Windows XP extends driver verification to catch more subtle bugs, improves the facilities for catching programming errors in user-level code, and subjects third-party applications, drivers, and devices to a rigorous certification process. Furthermore, Windows XP adds new facilities for monitoring the health of the PC, including downloading fixes for problems before they are encountered by users. The perceived reliability of Windows XP was also improved by making the graphical user interface easier to use through better visual design, simpler menus, and measured improvements in the ease with which users can discover how to perform common tasks.

22.2.3 Windows and POSIX Application Compatibility

Windows XP is not only an update of Windows 2000; it is a replacement for Windows 95/98. Windows 2000 focused primarily on compatibility for business applications. The requirements for Windows XP include a much higher compatibility with consumer applications that run on Windows 95/98. **Application compatibility** is difficult to achieve because each application checks for a particular version of Windows, may have some dependence on the

quirks of the implementation of APIs, may have latent application bugs that were masked in the previous system, and so forth.

Windows XP introduces a compatibility layer that falls between applications and the Win32 APIs. This layer makes Windows XP look (almost) bug-for-bug compatible with previous versions of Windows. Windows XP, like earlier NT releases, maintains support for running many 16-bit applications using a *thunking*, or conversion, layer that translates 16-bit API calls into equivalent 32-bit calls. Similarly, the 64-bit version of Windows XP provides a thunking layer that translates 32-bit API calls into native 64-bit calls. POSIX support in Windows XP is much improved. A new POSIX subsystem called Interix is now available. Most available UNIX-compatible software compiles and runs under Interix without modification.

22.2.4 High Performance

Windows XP is designed to provide **high performance** on desktop systems (which are largely constrained by I/O performance), server systems (where the CPU is often the bottleneck), and large multithreaded and multiprocessor environments (where locking and cache-line management are key to scalability). High performance has been an increasingly important goal for Windows XP. Windows 2000 with SQL 2000 on Compaq hardware achieved top TPC-C numbers at the time it shipped.

To satisfy performance requirements, NT uses a variety of techniques, such as asynchronous I/O, optimized protocols for networks (for example, optimistic locking of distributed data, batching of requests), kernel-based graphics, and sophisticated caching of file-system data. The memory-management and synchronization algorithms are designed with an awareness of the performance considerations related to cache lines and multiprocessors.

Windows XP has further improved performance by reducing the code-path length in critical functions, using better algorithms and per-processor data structures, using memory coloring for NUMA (non-uniform memory access) machines, and implementing more scalable locking protocols, such as queued spinlocks. The new locking protocols help reduce system bus cycles and include lock-free lists and queues, use of atomic read–modify–write operations (like interlocked increment), and other advanced locking techniques.

The subsystems that constitute Windows XP communicate with one another efficiently by a local procedure call (LPC) facility that provides high-performance message passing. Except while executing in the kernel dispatcher, threads in the subsystems of Windows XP can be preempted by higher-priority threads. Thus, the system responds quickly to external events. In addition, Windows XP is designed for symmetrical multiprocessing; on a multiprocessor computer, several threads can run at the same time.

22.2.5 Extensibility

Extensibility refers to the capacity of an operating system to keep up with advances in computing technology. So that changes over time are facilitated, the developers implemented Windows XP using a layered architecture. The Windows XP executive runs in kernel or protected mode and provides the basic system services. On top of the executive, several server subsystems operate in user mode. Among them are **environmental subsystems** that emulate

different operating systems. Thus, programs written for MS-DOS, Microsoft Windows, and POSIX all run on Windows XP in the appropriate environment. (See Section 22.4 for more information on environmental subsystems.) Because of the modular structure, additional environmental subsystems can be added without affecting the executive. In addition, Windows XP uses loadable drivers in the I/O system, so new file systems, new kinds of I/O devices, and new kinds of networking can be added while the system is running. Windows XP uses a client–server model like the Mach operating system and supports distributed processing by remote procedure calls (RPCs) as defined by the Open Software Foundation.

22.2.6 Portability

An operating system is **portable** if it can be moved from one hardware architecture to another with relatively few changes. Windows XP is designed to be portable. As is true of the UNIX operating system, the majority of the system is written in C and C++. Most processor-dependent code is isolated in a dynamic link library (DLL) called the **hardware-abstraction layer** (**HAL**). A DLL is a file that is mapped into a process's address space such that any functions in the DLL appear to be part of the process. The upper layers of the Windows XP kernel depend on the HAL interfaces rather than on the underlying hardware, bolstering Windows XP portability. The HAL manipulates hardware directly, isolating the rest of Windows XP from hardware differences among the platforms on which it runs.

Although for market reasons Windows 2000 shipped only on Intel IA32-compatible platforms, it was also tested on IA32 and DEC Alpha platforms until just prior to release to ensure portability. Windows XP runs on IA32-compatible and IA64 processors. Microsoft recognizes the importance of multiplatform development and testing, since, as a practical matter, maintaining portability is a matter of *use it or lose it*.

22.2.7 International Support

Windows XP is also designed for **international** and **multinational** use. It provides support for different locales via the **national-language-support (NLS) API**. The NLS API provides specialized routines to format dates, time, and money in accordance with various national customs. String comparisons are specialized to account for varying character sets. UNICODE is Windows XP's native character code. Windows XP supports ANSI characters by converting them to UNICODE characters before manipulating them (8-bit to 16-bit conversion). System text strings are kept in resource files that can be replaced to localize the system for different languages. Multiple locales can be used concurrently, which is important to multilingual individuals and businesses.

22.3 System Components

The architecture of Windows XP is a layered system of modules, as shown in Figure 22.1. The main layers are the HAL, the kernel, and the executive, all of which run in protected mode, and a collection of subsystems and services that run in user mode. The user-mode subsystems fall into two categories: