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 based on a model similar to that of 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 Q.
- **Exclusive locks:** When a transaction needs to lock data item Q, it requests a lock on Q from the lock manager at all sites containing a replica of Q.

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 is 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

In the case of data replication, we may 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 Q. As before, the response to the request is delayed until the request can be granted.

Thus, the primary copy enables concurrency control for replicated data to be handled in a manner similar to that for unreplicated data. This method of handling allows for a simple implementation. 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 that is used in deciding the serialization order. Our first task, then, in generalizing the centralized scheme to a distributed scheme is to develop a scheme for generating unique timestamps. Once this scheme has been developed, our previous protocols can be applied directly to the nonreplicated environment.

18.4.2.1 Generation of Unique Timestamps

There are two primary methods for generating unique timestamps, one centralized and one 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 unique local timestamp using either a logical counter or the local clock. The global unique timestamp is obtained by concatenation of the unique local 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 that of other sites. Therefore, all timestamps generated by the fast site will be larger than those generated by other sites. What is needed is a mechanism 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 unique local timestamp. The logical clock can be implemented as a counter that is

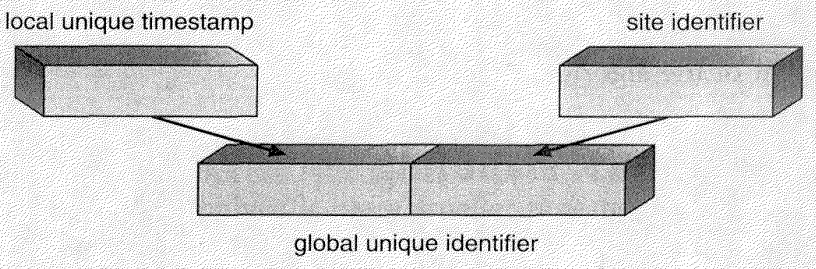

Figure 18.2 Generation of unique timestamps.

incremented after a new local timestamp is generated. 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 $<x,y>$ 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 $TS(T_j) < TS(T_i)$. Similarly, a **write**(x) operation by T_i must be delayed if there exists a transaction T_j that will perform either **read**(x) or **write**(x) operation and $TS(T_j) < TS(T_i)$. There are various methods for ensuring this property. One such method, called the *conservative timestamp-ordering scheme*, requires each site to maintain a **read** and **write** queue consisting of all the **read** and **write** requests, respectively, 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. Rather, 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 also be used in a distributed system. In the following, we describe several of these distributed algorithms.

18.5.1 Deadlock Prevention

The deadlock-prevention and deadlock-avoidance algorithms presented in Chapter 7 can also 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 channeled through the banker.

These two schemes can be used in dealing with the deadlock problem in a distributed environment. The first scheme is simple to implement and requires little overhead. The second scheme 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.

In this section, we present a new deadlock-prevention scheme that is based on a timestamp-ordering approach with resource preemption. 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 priority 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:

- **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.

- **The wound–wait scheme:** This approach is based on a preemptive technique and is a counterpart to the wait–die system. 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 *wound* 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 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 requested a resource it holds. When P_i is restarted and requests the resource now being held by P_j, P_i waits. Thus, there are fewer rollbacks in the wound–wait scheme.

The major problem with these two 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 elaborate this problem by describing several common techniques to deal with this issue. These schemes require that each site keep a *local* wait-for graph. The nodes of the graph correspond

to all the processes (local as well as nonlocal) that are currently either 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 A needs a resource held by process P_j in site B, a request message is sent by P_i to site B. The edge $P_i \rightarrow P_j$ is then inserted in the local wait-for graph of site B.

Clearly, if any local wait-for graph has a cycle, deadlock has occurred. On the other hand, the fact that there are no cycles in any of the local wait-for graphs does not mean that there are no deadlocks. 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 (shown in 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 deadlock state.

There are a number of different methods for organizing the wait-for graph in a distributed system. We shall describe several common schemes.

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. Obviously, the constructed graph must be generated such that, whenever the detection algorithm is invoked, the

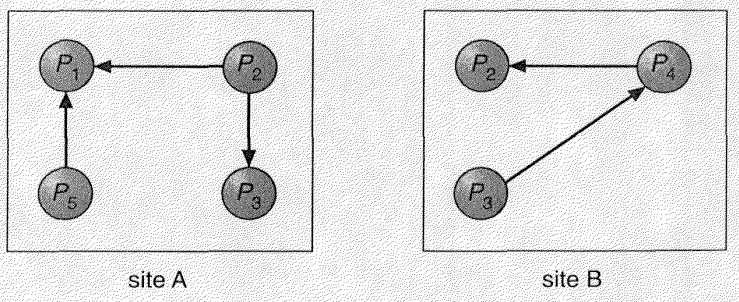

Figure 18.3 Two local wait-for graphs.

reported results are correct in a sense that, if a deadlock exists, then it is reported properly, and if a deadlock is reported, then the system is indeed in a deadlock state. As we shall show, it is not easy to construct such correct algorithms.

There are three different options (points in time) when the wait-for graph may be constructed:

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

Let us consider option 1. Whenever an edge is either inserted or removed in 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, 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 B 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 A 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.

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.

Note that, in this scheme (option 1), unnecessary rollbacks may occur, as a result of two situations:

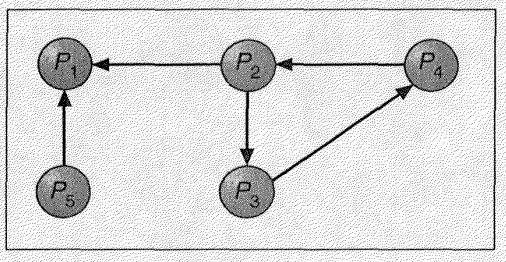

Figure 18.4 Global wait-for graph for Figure 18.3.

- *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 A, resulting in the deletion of the edge $P_1 \rightarrow P_2$ in A. Process P_2 then requests a resource held by P_3 at site B, resulting in the addition of the edge $P_2 \rightarrow P_3$ in B. If the *insert* $P_2 \rightarrow P_3$ message from B arrives before the *delete* $P_1 \rightarrow P_2$ message from A, 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.
- 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 was aborted for reasons unrelated to the deadlock (such as the process exceeding its allocated time). For example, suppose that site A 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.

Note that the same problems are inherited in solutions employing the other two options (that is, options 2 and 3).

Let us now present a centralized deadlock-detection algorithm, using option 3, which 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 (timestamps). When process P_i, at site A, requests a resource from P_j, at site B, 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 of A. This edge is inserted in the local wait-for graph of B only if B 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 then as follows:

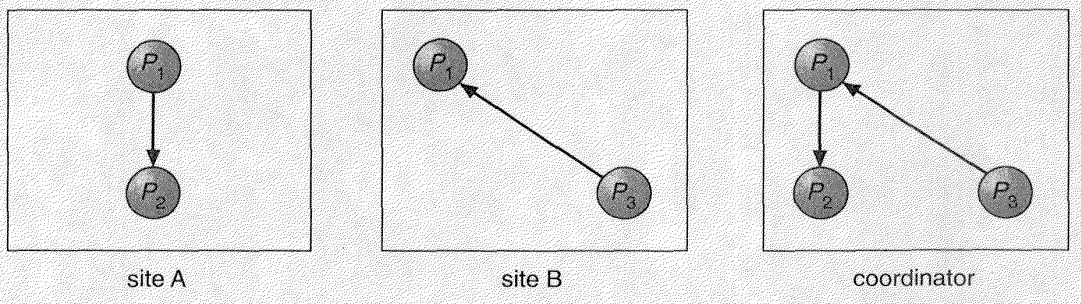

Figure 18.5 Local and global wait-for graphs.

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. Note that 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 (1) there is an edge $P_i \rightarrow P_j$ in one of the wait-for graphs, or (2) an edge $P_i \rightarrow P_j$ with some label TS appears in more than one wait-for graph.

We assert that, if there is a cycle in the constructed graph, then the system is in a deadlock state. If there is no cycle in the constructed graph, then the system was not in a deadlock 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. In this scheme, 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 there exists a process at another site that is waiting to acquire a resource currently being held by P_j in this local site.

To illustrate this situation, we consider 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 deadlock state. If, however, there exists a cycle involving P_{ex}, then this implies that there is a *possibility* of a deadlock. To

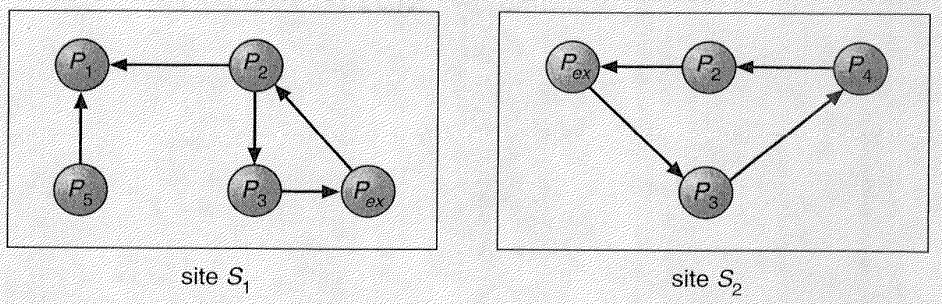

Figure 18.6 Augmented local wait-for graphs of Figure 18.3.

ascertain whether a deadlock does exist, we must invoke a distributed deadlock-detection algorithm.

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

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

which indicates that transaction 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 deadlock 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 transaction P_i a unique identifier, which we denote by $ID(P_i)$. When site S_k discovers that its local wait-for graph contains a cycle involving node P_{ex} of the form

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

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.

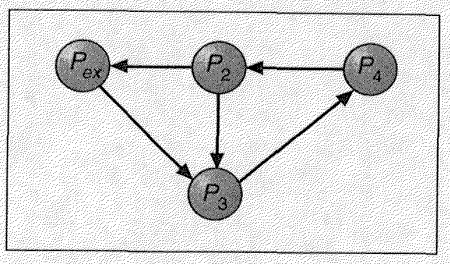

Figure 18.7 Augmented local wait-for graph in site S_2 of Figure 18.6.

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 deadlock 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, or 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 two interesting examples of election algorithms for two different configurations of distributed systems. The first algorithm is applicable to systems where every process can send a message to every other process in the system. The second algorithm is applicable 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 indeed has 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. (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 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.

Let us 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 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 (which P_1 does not receive, 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, using 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 Ring Algorithm

The *ring* algorithm assumes that the links are unidirectional, and that processes send their messages to their right neighbors. 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 would be to require 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." There are several reasons why such an agreement may not take place. 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.

This problem has been expressed as the *Byzantine generals problem.* Several divisions of the Byzantine army, each commanded by its own general, surround an enemy camp. The Byzantine generals must reach a common 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. There are at least two major reasons why the generals may not be able to reach an agreement:

- 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.
- 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 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 A, which has sent a message to process P_j at site B, 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 S if P_j has received its message, or to compute a function F 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 16.4.1. 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 to wait for an acknowledgment. This procedure continues until P_i either gets the acknowledgment message back, or is notified by the system that site B 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, to decide on how to proceed with its computation. For example, P_j may want to compute S only if it is assured that P_i got its acknowledgment. In other words, P_i and P_j will compute S 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.

Let us prove this claim. Suppose that there exists a minimal sequence of message transfers such that, after the messages have been delivered, both processes agree to compute S. 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 S 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 S.

18.7.2 Faulty Processes

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 $X_i = A_{i,1}, A_{i,2}, \ldots, A_{i,n})$ such that

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. These solutions 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 that value was sent by that 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 distributed systems. Timestamps can be used to provide a consistent event ordering in a distributed system.

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 network, is the token-passing approach.

For atomicity to be ensured, all the sites in which a transaction T 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, the only change that needs to be incorporated is in the way that the lock manager is implemented. In the case of timestamping and validation schemes, the only needed change 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. Different methods for organizing the wait-for graph include a centralized approach and a fully distributed approach.

Some of the distributed algorithms require the use of a coordinator. If the coordinator fails owing 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. It does 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 a 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 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.

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

18.4 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.

18.5 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 nonleaf 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 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.

18.6 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?

18.7 Consider a failure that occurs during two-phase commit for a transaction. For each possible failure, explain how two-phase commit ensures transaction atomicity despite the failure.

Bibliographic Notes

The distributed algorithm for extending the happened-before relation to a consistent total ordering of all the events in the system (Section 18.1) was developed by Lamport [1978a].

The first general algorithm for implementing mutual exclusion in a distributed environment was developed by Lamport [1978a] also. 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 was presented in Section 18.2.2. A square root algorithm for distributed mutual exclusion was presented by Maekawa [1985]. The token-passing algorithm for ring-structured systems presented in Section 18.2.3 was developed by Le Lann [1977]. Discussions concerning mutual exclusion in computer networks

were presented by Carvalho and Roucairol [1983]. An efficient and fault-tolerant solution of distributed mutual exclusion was presented by Agrawal and El Abbadi [1991]. 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 [1978a, 1978b], 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 assumption 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 covering 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 18.3 appeared in Menasce and Muntz [1979]. A survey of deadlock detection in distributed systems is offered by Knapp [1987] and Singhal [1989].

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]. The election algorithm for a ring-structured system was written by Le Lann [1977].

PART SIX

CASE STUDIES

The various concepts described in this book can now be drawn together by describing real operating systems. Two UNIX-based operating systems are covered in detail — Berkeley 4.3BSD and Mach. These operating systems were chosen in part because UNIX at one time was almost small enough to understand and yet is not a toy operating system. Most of its internal algorithms were selected for *simplicity*, not for speed or sophistication. UNIX is readily available to departments of computer science, so many students may have access to it. Mach gives us an opportunity to study a modern operating system that provides compatibility with 4.3BSD but has a vastly different design and implementation.

In addition to Berkeley 4.3BSD and Mach, we briefly discuss several other highly influential operating systems. The order of presentation has been chosen to highlight the similarities and differences of the systems; it is not strictly chronological, and does not reflect the relative importance of the system.

CHAPTER 19

THE UNIX SYSTEM

Although operating-system concepts can be considered in purely theoretical terms, it is often useful to see how they are implemented in practice. This chapter presents an in-depth examination of the 4.3BSD operating system, a version of UNIX, as an example of the various concepts presented in this book. By examining a complete, real system, we can see how the various concepts discussed in this book relate both to one another and to practice. We consider first a brief history of UNIX, and present the system's user and programmer interfaces. Then, we discuss the internal data structures and algorithms used by the UNIX kernel to support the user–programmer interface.

19.1 ■ History

The first version of UNIX was developed in 1969 by Ken Thompson of the Research Group at Bell Laboratories to use an otherwise idle PDP-7. He was soon joined by Dennis Ritchie. Thompson, Ritchie, and other members of the Research Group produced the early versions of UNIX.

Ritchie had previously worked on the MULTICS project, and MULTICS had a strong influence on the newer operating system. Even the name *UNIX* is merely a pun on *MULTICS*. The basic organization of the file system, the idea of the command interpreter (the shell) as a user process, the use of a separate process for each command, the original line-editing characters (# to erase the last character and @ to erase the entire line), and numerous other features came directly from MULTICS. Ideas from various other

operating systems, such as from MIT's CTSS and the XDS-940 system, were also used.

Ritchie and Thompson worked quietly on UNIX for many years. Their work on the first version allowed them to move it to a PDP-11/20, for a second version. A third version resulted from their rewriting most of the operating system in the systems-programming language C, instead of the previously used assembly language. C was developed at Bell Laboratories to support UNIX. UNIX was also moved to larger PDP-11 models, such as the 11/45 and 11/70. Multiprogramming and other enhancements were added when it was rewritten in C and moved to systems (such as the 11/45) that had hardware support for multiprogramming.

As UNIX developed, it became widely used within Bell Laboratories and gradually spread to a few universities. The first version widely available outside Bell Laboratories was Version 6, released in 1976. (The version number for early UNIX systems corresponds to the edition number of the *UNIX Programmer's Manual* that was current when the distribution was made; the code and the manuals were revised independently.)

In 1978, Version 7 was distributed. This UNIX system ran on the PDP-11/70 and the Interdata 8/32, and is the ancestor of most modern UNIX systems. In particular, it was soon ported to other PDP-11 models and to the VAX computer line. The version available on the VAX was known as 32V. Research has continued since then.

After the distribution of Version 7 in 1978, the UNIX Support Group (USG) assumed administrative control and responsibility from the Research Group for distributions of UNIX within AT&T, the parent organization for Bell Laboratories. UNIX was becoming a product, rather than simply a research tool. The Research Group has continued to develop their own version of UNIX, however, to support their own internal computing. Next came Version 8, which included a facility called the *stream I/O system* that allows flexible configuration of kernel IPC modules. It also contained RFS, a remote file system similar to Sun's NFS. Next came Versions 9 and 10 (the latter version, released in 1989, is available only within Bell Laboratories).

USG mainly provided support for UNIX within AT&T. The first external distribution from USG was System III, in 1982. System III incorporated features of Version 7, and 32V, and also of several UNIX systems developed by groups other than Research. Features of UNIX/RT, a real-time UNIX system, as well as numerous portions of the Programmer's Work Bench (PWB) software tools package were included in System III.

USG released System V in 1983; it is largely derived from System III. The divestiture of the various Bell operating companies from AT&T has left AT&T in a position to market System V aggressively. USG was restructured as the UNIX System Development Laboratory (USDL), which released UNIX System V Release 2 (V.2) in 1984. UNIX System V Release 2, Version 4 (V.2.4) added a new implementation of virtual memory with copy-on-write paging and shared memory. USDL was in turn replaced by AT&T

Information Systems (ATTIS), which distributed System V Release 3 (V.3) in 1987. V.3 adapts the V8 implementation of the stream I/O system and makes it available as *STREAMS*. It also includes RFS, an NFS-like remote file system.

The small size, modularity, and clean design of early UNIX systems led to UNIX-based work at numerous other computer-science organizations, such as at Rand, BBN, the University of Illinois, Harvard, Purdue, and even DEC. The most influential of the non-Bell Laboratories and non–AT&T UNIX development groups, however, has been the University of California at Berkeley.

The first Berkeley VAX UNIX work was the addition in 1978 of virtual memory, demand paging, and page replacement to 32V by Bill Joy and Ozalp Babaoglu to produce 3BSD UNIX. This version was the first implementation of any of these facilities on any UNIX system. The large virtual-memory space of 3BSD allowed the development of very large programs, such as Berkeley's own Franz LISP. The memory-management work convinced the Defense Advanced Research Projects Agency (DARPA) to fund Berkeley for the development of a standard UNIX system for government use; 4BSD UNIX was the result.

The 4BSD work for DARPA was guided by a steering committee that included many notable people from the UNIX and networking communities. One of the goals of this project was to provide support for the DARPA Internet networking protocols (TCP/IP). This support was provided in a general manner. It is possible in 4.2BSD to communicate uniformly among diverse network facilities, including local-area networks (such as Ethernets and token rings) and wide-area networks (such as NSFNET). This implementation was the most important reason for the current popularity of these protocols. It was used as the basis for the implementations of many vendors of UNIX computer systems, and even other operating systems. It permitted the Internet to grow from 60 connected networks in 1984 to more than 8000 networks and an estimated 10 million users in 1993.

In addition, Berkeley adapted many features from contemporary operating systems to improve the design and implementation of UNIX. Many of the terminal line-editing functions of the TENEX (TOPS-20) operating system were provided by a new terminal driver. A new user interface (the C Shell), a new text editor (ex/vi), compilers for Pascal and LISP, and many new systems programs were written at Berkeley. For 4.2BSD, certain efficiency improvements were inspired by the VMS operating system.

UNIX software from Berkeley is released in *Berkeley Software Distributions*. It is convenient to refer to the Berkeley VAX UNIX systems following 3BSD as 4BSD, although there were actually several specific releases, most notably 4.1BSD and 4.2BSD. The generic numbers BSD and 4BSD are used for the PDP-11 and VAX distributions of Berkeley UNIX. 4.2BSD, first

distributed in 1983, was the culmination of the original Berkeley DARPA UNIX project. 2.9BSD is the equivalent version for PDP-11 systems.

In 1986, 4.3BSD was released. It was so similar to 4.2BSD that its manuals described 4.2BSD more comprehensively than the 4.2BSD manuals did. It did include numerous internal changes, however, including bug fixes and performance improvements. Some new facilities also were added, including support for the Xerox Network System protocols.

4.3BSD Tahoe was the next version, released in 1988. It included various new developments, such as improved networking congestion control and TCP/IP performance. Also, disk configurations were separated from the device drivers, and are now read off the disks themselves. Expanded time-zone support is also included. 4.3BSD Tahoe was actually developed on and for the CCI Tahoe system (Computer Console, Inc., Power 6 computer), rather than for the usual VAX base. The corresponding PDP-11 release is 2.10.1BSD, which is distributed by the USENIX Association, which also publishes the 4.3BSD manuals. The 4.32BSD Reno release saw the inclusion of an implementation of ISO/OSI networking.

The last Berkeley release, 4.4BSD, was finalized in June of 1993. It includes new X.25 networking support, and POSIX standard compliance. It also has a radically new file system organization, with a new virtual file system interface and support for *stackable* file systems, allowing file systems to be layered on top of each other for easy inclusion of new features. An implementation of NFS is also included in the release (Chapter 17), as is a new log-based file system (see Chapter 12). The 4.4BSD virtual memory system is derived from Mach (described in the next chapter). Several other changes, such as enhanced security and improved kernel structure, are also included. With the release of version 4.4, Berkeley has halted its research efforts.

4BSD was the operating system of choice for VAXes from its initial release (1979) until the release of Ultrix, DEC's BSD implementation. 4BSD is still the best choice for many research and networking installations. Many organizations would buy a 32V license and order 4BSD from Berkeley without even bothering to get a 32V tape.

The current set of UNIX systems is not limited to those by Bell Laboratories, AT&T, and Berkeley, however. Sun Microsystems helped popularize the BSD flavor of UNIX by shipping it on their workstations. As UNIX has grown in popularity, it has been moved to many different computers and computer systems. A wide variety of UNIX, and UNIX-like, operating systems have been created. DEC supports its UNIX (called Ultrix) on its workstations and is replacing Ultrix with another UNIX-derived operating system, OSF/1; Microsoft rewrote UNIX for the Intel 8088 family and called it XENIX, and its new Windows/NT operating system is heavily influenced by UNIX; IBM has UNIX (AIX) on its PCs, workstations, and mainframes. In fact, UNIX is available on almost all general-purpose computers; it runs on personal computers, workstations, minicomputers,

mainframes, and supercomputers, from Apple Macintosh IIs to Cray IIs. Because of its wide availability, it is used in environments ranging from academic to military to manufacturing process control. Most of these systems are based on Version 7, System III, 4.2BSD, or System V.

The wide popularity of UNIX with computer vendors has made UNIX the most portable of operating systems, and has made it possible for users to expect a UNIX environment independent of any specific computer manufacturer. But the large number of implementations of the system has led to remarkable variation in the programming and user interfaces distributed by the vendors. For true vendor independence, application-program developers need consistent interfaces. Such interfaces would allow all "UNIX" applications to run on all UNIX systems, which is certainly not the current situation. This issue has become important as UNIX has become the preferred program-development platform for applications ranging from databases to graphics and networking, and has led to a strong market demand for UNIX standards.

There are several standardization projects underway, starting with the */usr/group 1984 Standard* sponsored by the UniForum industry user's group. Since then, many official standards bodies have continued the effort, including IEEE and ISO (the POSIX standard). The X/Open Group international consortium completed XPG3, a Common Application Environment, which subsumes the IEEE interface standard. Unfortunately, XPG3 is based on a draft of the ANSI C standard, rather than the final specification, and therefore needs to be redone. The XPG4 is due out in 1993. In 1989, the ANSI standards body standardized the C programming language, producing an ANSI C specification that vendors were quick to adopt. As these projects continue, the variant flavors of UNIX will converge and there will be one programming interface to UNIX, allowing UNIX to become even more popular. There are in fact two separate sets of powerful UNIX vendors working on this problem: the AT&T-guided UNIX International (UI) and the Open Software Foundation (OSF) have both agreed to follow the POSIX standard. Recently, many of the vendors involved in those two groups have agreed on further standardization (the COSE agreement) on the Motif window environment, and ONC+ (which includes Sun RPC and NFS) and DCE network facilities (which includes AFS and an RPC package).

AT&T replaced its ATTIS group in 1989 with the UNIX Software Organization (USO), which shipped the first merged UNIX, System V Release 4. This system combines features from System V, 4.3BSD, and Sun's SunOS, including long file names, the Berkeley file system, virtual memory management, symbolic links, multiple access groups, job control, and reliable signals; it also conforms to the published POSIX standard, POSIX.1. After USO produced SVR4, it became an independent AT&T subsidiary named Unix System Laboratories (USL); in 1993, it was purchased by Novell, Inc.

Figure 19.1 summarizes the relationships among the various versions of UNIX.

The UNIX system has grown from a personal project of two Bell Laboratories employees to an operating system being defined by multinational standardization bodies. Yet this system is still of interest to academia. We believe that UNIX has become and will remain an important

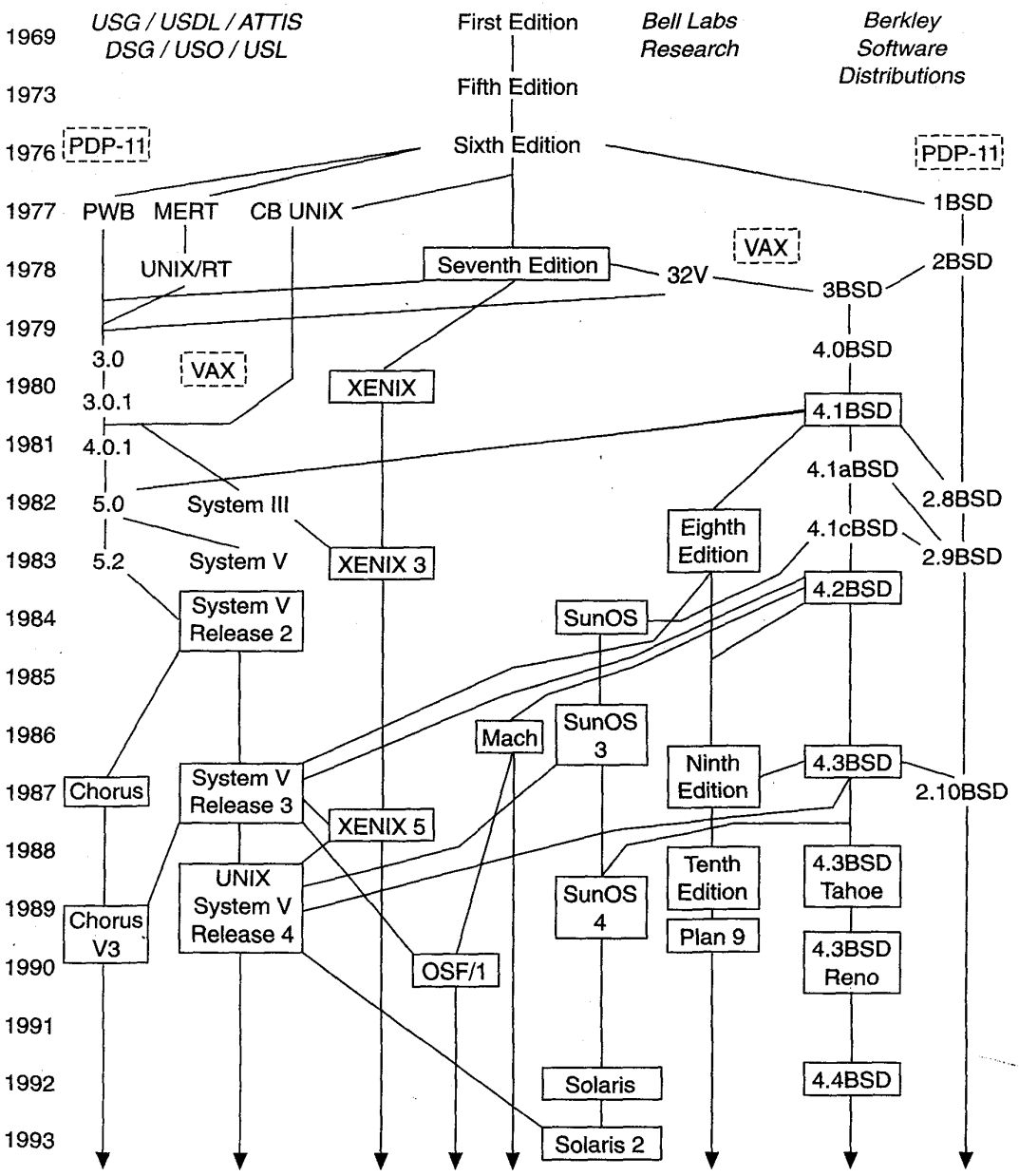

Figure 19.1 History of UNIX versions.

part of operating-system theory and practice. UNIX is an excellent vehicle for academic study. For example, the Tunis operating system, the Xinu operating system, and the Minix operating system are based on the concepts of UNIX, but were developed explicitly for classroom study. There is a plethora of ongoing UNIX-related research systems, including Mach, Chorus, Comandos, and Roisin. The original developers, Ritchie and Thompson, were honored in 1983 by the Association for Computing Machinery Turing award for their work on UNIX.

The specific UNIX version used in this chapter is the VAX version of 4.3BSD. This system is used because it implements many interesting operating-system concepts, such as demand paging with clustering, and networking. It has also been influential in other UNIX systems, in standards, and in networking developments. The VAX implementation is used because 4.3BSD was developed on the VAX and that machine still represents a convenient point of reference, despite the recent proliferation of implementations on other hardware (such as the Motorola 68040 and 88000, the Intel i486, the Sun SPARC, DEC Alpha, HP Precision, and the MIPS R4000 CPUs).

19.2 ■ Design Principles

UNIX was designed to be a time-sharing system. The standard user interface (the shell) is simple and can be replaced by another, if desired. The file system is a multilevel tree, which allows users to create their own subdirectories. Each user data file is simply a sequence of bytes.

Disk files and I/O devices are treated as similarly as possible. Thus, device dependencies and peculiarities are kept in the kernel as much as possible; even in the kernel, most of them are confined to the device drivers.

UNIX supports multiple processes. A process can easily create new processes. CPU scheduling is a simple priority algorithm. 4.3BSD uses demand paging as a mechanism to support memory-management and CPU-scheduling decisions. Swapping is used if a system is suffering from excess paging.

Because UNIX was originated first by one programmer, Ken Thompson, and then by another, Dennis Ritchie, as a system for their own convenience, it was small enough to understand. Most of the algorithms were selected for *simplicity*, not for speed or sophistication. The intent was to have the kernel and libraries provide a small set of facilities that was sufficiently powerful to allow a person to build a more complex system if one were needed. UNIX's clean design has resulted in many imitations and modifications.

Although the designers of UNIX had a significant amount of knowledge about other operating systems, UNIX had no elaborate design spelled out

before its implementation. This flexibility appears to have been one of the key factors in the development of the system. Some design principles were involved, however, even though they were not made explicit at the outset.

The UNIX system was designed by programmers for programmers. Thus, it has always been interactive, and facilities for program development have always been a high priority. Such facilities include the program *make* (which can be used to check to see which of a collection of source files for a program need to be compiled, and then to do the compiling) and the *Source Code Control System (SCCS)* (which is used to keep successive versions of files available without having to store the entire contents of each step).

The operating system is written mostly in C, which was developed to support UNIX, since neither Thompson nor Ritchie enjoyed programming in assembly language. The avoidance of assembly language was also necessary because of the uncertainty about the machine or machines on which UNIX would be run. It has greatly simplified the problems of moving UNIX from one hardware system to another.

From the beginning, UNIX development systems have had all the UNIX sources available on-line, and the developers have used the systems under development as their primary systems. This pattern of development has greatly facilitated the discovery of deficiencies and their fixes, as well as of new possibilities and their implementations. It has also encouraged the plethora of UNIX variants existing today, but the benefits have outweighed the disadvantages: if something is broken, it can be fixed at a local site; there is no need to wait for the next release of the system. Such fixes, as well as new facilities, may be incorporated into later distributions.

The size constraints of the PDP-11 (and earlier computers used for UNIX) have forced a certain elegance. Where other systems have elaborate algorithms for dealing with pathological conditions, UNIX just does a controlled crash called *panic*. Instead of attempting to cure such conditions, UNIX tries to prevent them. Where other systems would use brute force or macro-expansion, UNIX mostly has had to develop more subtle, or at least simpler, approaches.

These early strengths of UNIX produced much of its popularity, which in turn produced new demands that challenged those strengths. UNIX was used for tasks such as networking, graphics, and real-time operation, which did not always fit into its original text-oriented model. Thus, changes were made to certain internal facilities and new programming interfaces were added. These new facilities, and others — particularly window interfaces — required large amounts of code to support them, radically increasing the size of the system. For instance, networking and windowing both doubled the size of the system. This pattern in turn pointed out the continued strength of UNIX — whenever a new development occurred in the industry, UNIX could usually absorb it, but still remain UNIX.

19.3 ■ Programmer Interface

As do most computer systems, UNIX consists of two separable parts: the kernel and the systems programs. We can view the UNIX operating system as being layered, as shown in Figure 19.2. Everything below the system-call interface and above the physical hardware is the *kernel*. The kernel provides the file system, CPU scheduling, memory management, and other operating-system functions through system calls. Systems programs use the kernel-supported system calls to provide useful functions, such as compilation and file manipulation.

System calls define the *programmer interface* to UNIX; the set of systems programs commonly available defines the *user interface*. The programmer and user interface define the context that the kernel must support.

System calls in VAX 4.2BSD are made by a trap to location 40 of the VAX interrupt vectors. Parameters are passed to the kernel on the hardware stack; the kernel returns values in registers R0 and R1. Register R0 may also return an error code. The carry bit distinguishes a normal return from an error return.

This level of detail is seldom seen by a UNIX programmer, fortunately. Most systems programs are written in C, and the *UNIX Programmer's Manual* presents all system calls as C functions. A system program written in C for 4.3BSD on the VAX can generally be moved to another 4.3BSD system and simply recompiled, even though the two systems may be quite different. The details of system calls are known only to the compiler. This feature is a major reason for the portability of UNIX programs.

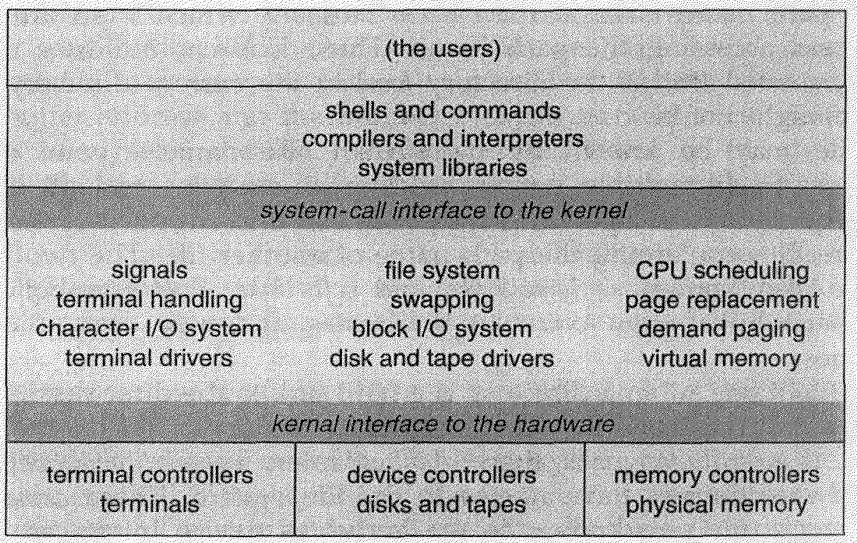

Figure 19.2 4.3BSD layer structure.

System calls for UNIX can be roughly grouped into three categories: file manipulation, process control, and information manipulation. In Chapter 3, we listed a fourth category, device manipulation, but since devices in UNIX are treated as (special) files, the same system calls support both files and devices (although there is an extra system call for setting device parameters).

19.3.1 File Manipulation

A *file* in UNIX is a sequence of bytes. Different programs expect various levels of structure, but the kernel does not impose a structure on files. For instance, the convention for text files is lines of ASCII characters separated by a single newline character (which is the linefeed character in ASCII), but the kernel knows nothing of this convention.

Files are organized in tree-structured *directories.* Directories are themselves files that contain information on how to find other files. A *path name* to a file is a text string that identifies a file by specifying a path through the directory structure to the file. Syntactically, it consists of individual file-name elements separated by the slash character. For example, in */usr/local/font,* the first slash indicates the root of the directory tree, called the *root* directory. The next element, *usr,* is a subdirectory of the root, *local* is a subdirectory of *usr,* and *font* is a file or directory in the directory *local.* Whether *font* is an ordinary file or a directory cannot be determined from the path-name syntax.

UNIX has both *absolute path names* and *relative path names.* Absolute path names start at the root of the file system and are distinguished by a slash at the beginning of the path name; */usr/local/font* is an absolute path name. Relative path names start at the *current directory,* which is an attribute of the process accessing the path name. Thus, *local/font* indicates a file or directory named *font* in the directory *local* in the current directory, which might or might not be */usr.*

A file may be known by more than one name in one or more directories. Such multiple names are known as *links,* and all links are treated equally by the operating system. 4.3BSD also supports *symbolic links,* which are files containing the path name of another file. The two kinds of links are also known as *hard links* and *soft links.* Soft (symbolic) links, unlike hard links, may point to directories and may cross file-system boundaries.

The file name "." in a directory is a hard link to the directory itself. The file name ".." is a hard link to the parent directory. Thus, if the current directory is */user/jlp/programs,* then *../bin/wdf* refers to */user/jlp/bin/wdf.*

Hardware devices have names in the file system. These *device special files* or *special files* are known to the kernel as device interfaces, but are nonetheless accessed by the user by much the same system calls as are other files.

Figure 19.3 shows a typical UNIX file system. The root (/) normally contains a small number of directories as well as */vmunix*, the binary boot image of the operating system; */dev* contains the device special files, such as */dev/console*, */dev/lp0*, */dev/mt0*, and so on; */bin* contains the binaries of the essential UNIX systems programs. Other binaries may be in */usr/bin* (for applications systems programs, such as text formatters), */usr/ucb* (for systems programs written by Berkeley rather than by AT&T), or */usr/local/bin* (for systems programs written at the local site). Library files — such as the C, Pascal, and FORTRAN subroutine libraries — are kept in */lib* (or */usr/lib* or */usr/local/lib*).

The files of users themselves are stored in a separate directory for each user, typically in */user*. Thus, the user directory for *carol* would normally be in */user/carol*. For a large system, these directories may be further grouped to ease administration, creating a file structure with */user/prof/avi* and */user/staff/carol*. Administrative files and programs, such as the password file, are kept in */etc*. Temporary files can be put in */tmp*, which is normally erased during system boot, or in */usr/tmp*.

Each of these directories may have considerably more structure. For example, the font-description tables for the troff formatter for the Merganthaler 202 typesetter are kept in */usr/lib/troff/dev202*. All the conventions concerning the location of specific files and directories have been defined by programmers and their programs; the operating-system kernel needs only */etc/init*, which is used to initialize terminal processes, to be operable.

System calls for basic file manipulation are **creat, open, read, write, close, unlink,** and **trunc**. The **creat** system call, given a path name, creates an (empty) file (or truncates an existing one). An existing file is opened by the **open** system call, which takes a path name and a mode (such as read, write, or read−write) and returns a small integer, called a *file descriptor*. A file descriptor may then be passed to a **read** or **write** system call (along with a buffer address and the number of bytes to transfer) to perform data transfers to or from the file. A file is closed when its file descriptor is passed to the **close** system call. The **trunc** call reduces the length of a file to 0.

A file descriptor is an index into a small table of open files for this process. Descriptors start at 0 and seldom get higher than 6 or 7 for typical programs, depending on the maximum number of simultaneously open files.

Each **read** or **write** updates the current offset into the file, which is associated with the file-table entry and is used to determine the position in the file for the next **read** or **write**. The **lseek** system call allows the position to be reset explicitly. It also allows the creation of sparse files (files with "holes" in them). The **dup** and **dup2** system calls can be used to produce a new file descriptor that is a copy of an existing one. The **fcntl** system call can also do that, and in addition can examine or set various parameters of

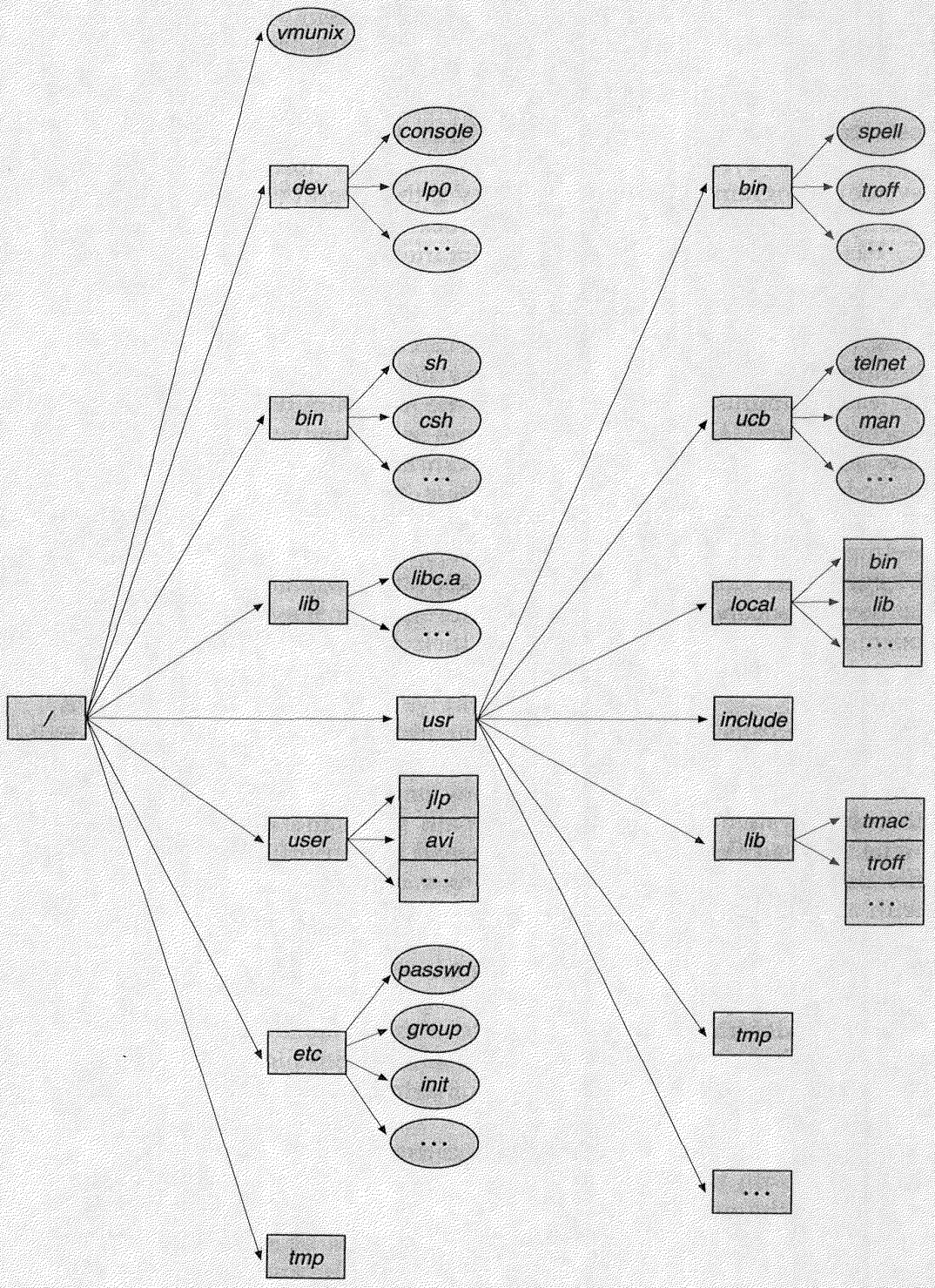

Figure 19.3 Typical UNIX directory structure.

an open file. For example, it can make each succeeding **write** to an open file append to the end of that file. There is an additional system call, **ioctl,** for manipulating device parameters. It can set the baud rate of a serial port, or rewind a tape, for instance.

Information about the file (such as its size, protection modes, owner, and so on) can be obtained by the **stat** system call. Several system calls allow some of this information to be changed: **rename** (change file name), **chmod** (change the protection mode), and **chown** (change the owner and group). Many of these system calls have variants that apply to file descriptors instead of file names. The **link** system call makes a hard link for an existing file, creating a new name for an existing file. A link is removed by the **unlink** system call; if it is the last link, the file is deleted. The **symlink** system call makes a symbolic link.

Directories are made by the **mkdir** system call and are deleted by **rmdir**. The current directory is changed by **cd**.

Although it is possible to use the standard file calls (**open** and others) on directories, it is inadvisable to do so, since directories have an internal structure that must be preserved. Instead, another set of system calls is provided to open a directory, to step through each file entry within the directory, to close the directory, and to perform other functions; these are **opendir, readdir, closedir,** and others.

19.3.2 Process Control

A *process* is a program in execution. Processes are identified by their *process identifier,* which is an integer. A new process is created by the **fork** system call. The new process consists of a copy of the address space of the original process (the same program and the same variables with the same values). Both processes (the parent and the child) continue execution at the instruction after the **fork** with one difference: The return code for the **fork** is zero for the new (child) process, whereas the (nonzero) process identifier of the child is returned to the parent.

Typically, the **execve** system call is used after a fork by one of the two processes to replace that process' virtual memory space with a new program. The **execve** system call loads a binary file into memory (destroying the memory image of the program containing the **execve** system call) and starts its execution.

A process may terminate by using the **exit** system call, and its parent process may wait for that event by using the **wait** system call. If the child process crashes, the system simulates the **exit** call. The **wait** system call provides the process id of a terminated child so that the parent can tell which of possibly many children terminated. A second system call, **wait3,** is similar to **wait** but also allows the parent to collect performance statistics about the child. Between the time the child exits, and the time the parent

completes one of the **wait** system calls, the child is *defunct*. A defunct process can do nothing, but exists merely so that the parent can collect its status information. If the parent process of a defunct process exits before a child, the defunct process is inherited by the *init* process (which in turn **wait**s on it) and becomes a *zombie* process. A typical use of these facilities is shown in Figure 19.4.

The simplest form of communication between processes is by *pipes*, which may be created before the **fork,** and whose endpoints are then set up between the **fork** and the **execve**. A pipe is essentially a queue of bytes between two processes. The pipe is accessed by a file descriptor, like an ordinary file. One process writes into the pipe, and the other reads from the pipe. The size of the original pipe system was fixed by the system. With 4.3BSD, pipes are implemented on top of the socket system, which has variable-sized buffers. Reading from an empty pipe or writing into a full pipe causes the process to be blocked until the state of the pipe changes. Special arrangements are needed for a pipe to be placed between a parent and child (so only one is reading and one is writing).

All user processes are descendants of one original process, called *init* (which has process identifier 1). Each terminal port available for interactive use has a *getty* process forked for it by *init*. The *getty* process initializes terminal line parameters and waits for a user's *login name*, which it passes through an **execve** as an argument to a *login* process. The *login* process collects the user's password, encrypts the password, and compares the result to an encrypted string taken from the file */etc/passwd*. If the comparison is successful, the user is allowed to log in. The *login* process executes a *shell*, or command interpreter, after setting the numeric *user identifier* of the process to that of the user logging in. (The shell and the user identifier are found in */etc/passwd* by the user's login name.) It is with this shell that the user ordinarily communicates for the rest of the login session; the shell itself forks subprocesses for the commands the user tells it to execute.

The user identifier is used by the kernel to determine the user's permissions for certain system calls, especially those involving file

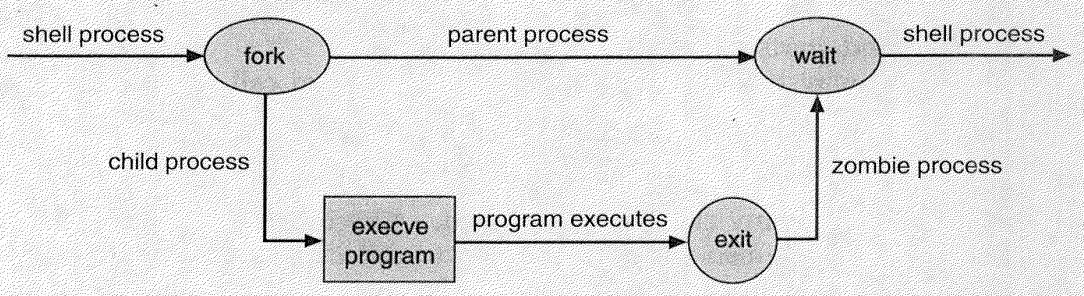

Figure 19.4 A shell forks a subprocess to execute a program.

accesses. There is also a *group identifier,* which is used to provide similar privileges to a collection of users. In 4.3BSD a process may be in several groups simultaneously. The *login* process puts the shell in all the groups permitted to the user by the files */etc/passwd* and */etc/group*.

There are actually two user identifiers used by the kernel: the *effective user identifier* is the identifier used to determine file access permissions. If the file of a program being loaded by an **execve** has the **setuid** bit set in its inode, the effective user identifier of the process is set to the user identifier of the owner of the file, whereas the *real user identifier* is left as it was. This scheme allows certain processes to have more than ordinary privileges while still being executable by ordinary users. The **setuid** idea was patented by Dennis Ritchie (U.S. Patent 4,135,240) and is one of the distinctive features of UNIX. There is a similar **setgid** bit for groups. A process may determine its real and effective user identifier with the **getuid** and **geteuid** calls, respectively. The **getgid** and **getegid** calls determine the process identifier and group identifier, respectively. The rest of a process' groups may be found with the **getgroups** system call.

19.3.3 Signals

Signals are a facility for handling exceptional conditions similar to software interrupts. There are 20 different signals, each corresponding to a distinct condition. A signal may be generated by a keyboard interrupt, by an error in a process (such as a bad memory reference), or by a number of asynchronous events (such as timers or job-control signals from the shell). Almost any signal may also be generated by the **kill** system call.

The *interrupt* signal, SIGINT, is used to stop a command before that command completes. It is usually produced by the ^C character (ASCII 3). As of 4.2BSD, the important keyboard characters are defined by a table for each terminal and can be redefined easily. The *quit* signal, SIGQUIT, is usually produced by the ^\ character (ASCII 28). The *quit* signal both stops the currently executing program and dumps its current memory image to a file named *core* in the current directory. The core file can be used by debuggers. SIGILL is produced by an illegal instruction and SIGSEGV by an attempt to address memory outside of the legal virtual-memory space of a process.

Arrangements can be made either for most signals to be ignored (to have no effect), or for a routine in the user process (a signal handler) to be called. A signal handler may safely do one of two things before returning from catching a signal: call the **exit** system call, or modify a global variable. There is one signal (the *kill* signal, number 9, SIGKILL) that cannot be ignored or caught by a signal handler. SIGKILL is used, for example, to kill a runaway process that is ignoring other signals such as SIGINT or SIGQUIT.

Signals can be lost: If another signal of the same kind is sent before a previous signal has been accepted by the process to which it is directed,

the first signal will be overwritten and only the last signal will be seen by the process. In other words, a call to the signal handler tells a process that there has been at least one occurance of the signal. Also, there is no relative priority among UNIX signals. If two different signals are sent to the same process at the same time, it is indeterminate which one the process will receive first.

Signals were originally intended to deal with exceptional events. As is true of the use of most other features in UNIX, however, signal use has steadily expanded. 4.1BSD introduced job control, which uses signals to start and stop subprocesses on demand. This facility allows one shell to control multiple processes: starting, stopping, and backgrounding them as the user wishes. 4.3BSD added the SIGWINCH signal, invented by Sun Microsystems, for informing a process that the window in which output is being displayed has changed size. Signals are also used to deliver urgent data from network connections.

Users also wanted more reliable signals, and a bug fix in an inherent race condition in the old signals implementation. Thus, 4.2BSD also brought with it a race-free, reliable, separately implemented signal capability. It allows individual signals to be blocked during critical sections, and has a new system call to let a process sleep until interrupted. It is similar to hardware-interrupt functionality. This capability is now part of the POSIX standard.

19.3.4 Process Groups

Groups of related processes frequently cooperate to accomplish a common task. For instance, processes may create, and communicate over, pipes. Such a set of processes is termed a *process group*, or a *job*. Signals may be sent to all processes in a group. A process usually inherits its process group from its parent, but the **setpgrp** system call allows a process to change its group.

Process groups are used by the C shell to control the operation of multiple jobs. Only one process group may use a terminal device for I/O at any time. This *foreground* job has the attention of the user on that terminal while all other nonattached jobs (*background* jobs) perform their function without user interaction. Access to the terminal is controlled by process group signals. Each job has a *controlling terminal* (again, inherited from its parent). If the process group of the controlling terminal matches the group of a process, that process is in the foreground, and is allowed to perform I/O. If a nonmatching (background) process attempts the same, a SIGTTIN or SIGTTOU signal is sent to its process group. This signal usually results in the process group freezing until it is foregrounded by the user, at which point it receives a SIGCONT signal, indicating that the process can perform the I/O. Similarly, a SIGSTOP may be sent to the foreground process group to freeze it.

19.3.5 Information Manipulation

System calls exist to set and return both an interval timer (**getitimer/setitimer**) and the current time (**gettimeofday/settimeofday**) in microseconds. In addition, processes can ask for their process identifier (**getpid**), their group identifier (**getgid**), the name of the machine on which they are executing (**gethostname**), and many other values.

19.3.6 Library Routines

The system-call interface to UNIX is supported and augmented by a large collection of library routines and header files. The header files provide the definition of complex data structures used in system calls. In addition, a large library of functions provides additional program support.

For example, the UNIX I/O system calls provide for the reading and writing of blocks of bytes. Some applications may want to read and write only 1 byte at a time. Although it would be possible to read and write 1 byte at a time, that would require a system call for each byte — a very high overhead. Instead, a set of standard library routines (the standard I/O package accessed through the header file *<stdio.h>*) provides another interface, which reads and writes several thousand bytes at a time using local buffers, and transfers between these buffers (in user memory) when I/O is desired. Formatted I/O is also supported by the standard I/O package.

Additional library support is provided for mathematical functions, network access, data conversion, and so on. The 4.3BSD kernel supports over 150 system calls; the C program library has over 300 library functions. Although the library functions eventually result in system calls where necessary (for example, the *getchar* library routine will result in a **read** system call if the file buffer is empty), it is generally unnecessary for the programmer to distinguish between the basic set of kernel system calls and the additional functions provided by library functions.

19.4 ■ User Interface

Both the programmer and the user of a UNIX system deal mainly with the set of systems programs that have been written and are available for execution. These programs make the necessary system calls to support their function, but the system calls themselves are contained within the program and do not need to be obvious to the user.

The common systems programs can be grouped into several categories; most of them are file or directory oriented. For example, the system programs to manipulate directories are *mkdir* to create a new directory, *rmdir* to remove a directory, *cd* to change the current directory to another, and *pwd* to print the absolute path name of the current (working) directory.

The *ls* program lists the names of the files in the current directory. Any of 18 options can ask that properties of the files be displayed also. For example, the *-l* option asks for a long listing, showing the file name, owner, protection, date and time of creation, and size. The *cp* program creates a new file that is a copy of an existing file. The *mv* program moves a file from one place to another in the directory tree. In most cases, this move simply requires a renaming of the file; if necessary, however, the file is copied to the new location and the old copy is deleted. A file is deleted by the *rm* program (which makes an **unlink** system call).

To display a file on the terminal, a user can run *cat*. The *cat* program takes a list of files and concatenates them, copying the result to the standard output, commonly the terminal. On a high-speed cathode-ray tube (CRT) display, of course, the file may speed by too fast to be read. The *more* program displays the file one screen at a time, pausing until the user types a character to continue to the next screen. The *head* program displays just the first few lines of a file; *tail* shows the last few lines.

These are the basic systems programs widely used in UNIX. In addition, there are a number of editors (*ed, sed, emacs, vi,* and so on), compilers (C, Pascal, FORTRAN, and so on), and text formatters (troff, TEX, scribe, and so on). There are also programs for sorting (*sort*) and comparing files (*cmp, diff*), looking for patterns (*grep, awk*), sending mail to other users (*mail*), and many other activities.

19.4.1 Shells and Commands

Both user-written and systems programs are normally executed by a command interpreter. The command interpreter in UNIX is a user process like any other. It is called a *shell,* as it surrounds the kernel of the operating system. Users can write their own shell, and there are, in fact, several shells in general use. The *Bourne shell,* written by Steve Bourne, is probably the most widely used — or, at least, it is the most widely available. The *C shell,* mostly the work of Bill Joy, a founder of Sun Microsystems, is the most popular on BSD systems. The Korn shell, by Dave Korn, has become popular because it combines the features of the Bourne shell and the C shell.

The common shells share much of their command-language syntax. UNIX is normally an interactive system. The shell indicates its readiness to accept another command by typing a prompt, and the user types a command on a single line. For instance, in the line

```
% ls -l
```

the percent sign is the usual C shell prompt, and the *ls -l* (typed by the user) is the (long) list-directory command. Commands can take arguments,

which the user types after the command name on the same line, separated by white space (spaces or tabs).

Although there are a few commands built into the shells (such as *cd*), a typical command is an executable binary object file. A list of several directories, the *search path*, is kept by the shell. For each command, each of the directories in the search path is searched, in order, for a file of the same name. If a file is found, it is loaded and executed. The search path can be set by the user. The directories */bin* and */usr/bin* are almost always in the search path, and a typical search path on a BSD system might be

(. /home/prof/avi/bin /usr/local/bin /usr/ucb /bin /usr/bin)

The *ls* command's object file is */bin/ls*, and the shell itself is */bin/sh* (the Bourne shell) or */bin/csh* (the C shell).

Execution of a command is done by a **fork** system call followed by an **execve** of the object file. The shell usually then does a **wait** to suspend its own execution until the command completes (Figure 19.4). There is a simple syntax (an ampersand [&] at the end of the command line) to indicate that the shell should *not* wait for the completion of the command. A command left running in this manner while the shell continues to interpret further commands is said to be a *background* command, or to be running in the background. Processes for which the shell *does* wait are said to run in the *foreground*.

The C shell in 4.3BSD systems provides a facility called *job control* (partially implemented in the kernel), as mentioned previously. Job control allows processes to be moved between the foreground and the background. The processes can be stopped and restarted on various conditions, such as a background job wanting input from the user's terminal. This scheme allows most of the control of processes provided by windowing or layering interfaces, but requires no special hardware. Job control is also useful in window systems, such as the X Window System developed at MIT. Each window is treated as a terminal, allowing multiple processes to be in the foreground (one per window) at any one time. Of course, background processes may exist on any of the windows. The Korn shell also supports job control, and it is likely that job control (and process groups) will be standard in future versions of UNIX.

19.4.2 Standard I/O

Processes can open files as they like, but most processes expect three file descriptors (numbers 0, 1, and 2) to be open when they start. These file descriptors are inherited across the **fork** (and possibly the **execve**) that created the process. They are known as *standard input* (0), *standard output* (1), and *standard error* (2). All three are frequently open to the user's

terminal. Thus, the program can read what the user types by reading standard input, and the program can send output to the user's screen by writing to standard output. The standard-error file descriptor is also open for writing and is used for error output; standard output is used for ordinary output. Most programs can also accept a file (rather than a terminal) for standard input and standard output. The program does not care where its input is coming from and where its output is going. This is one of the elegant design features of UNIX.

The common shells have a simple syntax for changing what files are open for the standard I/O streams of a process. Changing a standard file is called *I/O redirection*. The syntax for I/O redirection is shown in Figure 19.5. In this example, the *ls* command produces a listing of the names of files in the current directory, the *pr* command formats that list into pages suitable for a printer, and the *lpr* command spools the formatted output to a printer, such as */dev/lp0*. The subsequent command forces all output and all error messages to be redirected to a file. Without the ampersand, error messages appear on the terminal.

19.4.3 Pipelines, Filters, and Shell Scripts

The first three commands of Figure 19.5 could have been coalesced into the one command

% ls | pr | lpr

Each vertical bar tells the shell to arrange for the output of the preceding command to be passed as input to the following command. A pipe is used to carry the data from one process to the other. One process writes into one end of the pipe, and another process reads from the other end. In the example, the write end of one pipe would be set up by the shell to be the standard output of *ls*, and the read end of the pipe would be the standard input of *pr*; there would be another pipe between *pr* and *lpr*.

Command	Meaning of command
% ls > filea	direct output of *ls* to file *filea*
% pr < filea > fileb	input from *filea* and output to *fileb*
% lpr < fileb	input from *fileb*
%	
% make program >& errs	save both standard output and standard error in a file

Figure 19.5 Standard I/O redirection.

A command such as *pr* that passes its standard input to its standard output, performing some processing on it, is called a *filter*. Many UNIX commands can be used as filters. Complicated functions can be pieced together as pipelines of common commands. Also, common functions, such as output formatting, do not need to be built into numerous commands, because the output of almost any program can be piped through *pr* (or some other appropriate filter).

Both of the common UNIX shells are also programming languages, with shell variables and the usual higher-level programming-language control constructs (loops, conditionals). The execution of a command is analogous to a subroutine call. A file of shell commands, a *shell script*, can be executed like any other command, with the appropriate shell being invoked automatically to read it. *Shell programming* thus can be used to combine ordinary programs conveniently for sophisticated applications without the necessity of any programming in conventional languages.

This external user view is commonly thought of as the definition of UNIX, yet it is the most easily changed definition. Writing a new shell with a quite different syntax and semantics would greatly change the user view while not changing the kernel or even the programmer interface. Several menu-driven and iconic interfaces for UNIX now exist, and the X Window System is rapidly becoming a standard. The heart of UNIX is, of course, the kernel. This kernel is much more difficult to change than is the user interface, because all programs depend on the system calls that it provides to remain consistent. Of course, new system calls can be added to increase functionality, but programs must then be modified to use the new calls.

19.5 ■ Process Management

A major design problem for operating systems is the representation of processes. One substantial difference between UNIX and many other systems is the ease with which multiple processes can be created and manipulated. These processes are represented in UNIX by various control blocks. There are no system control blocks accessible in the virtual address space of a user process; control blocks associated with a process are stored in the kernel. The information in these control blocks is used by the kernel for process control and CPU scheduling.

19.5.1 Process Control Blocks

The most basic data structure associated with processes is the *process structure*. A process structure contains everything that the system needs to know about a process when the process is swapped out, such as its unique process identifier, scheduling information (such as the priority of the process), and pointers to other control blocks. There is an array of process

structures whose length is defined at system linking time. The process structures of ready processes are kept linked together by the scheduler in a doubly linked list (the ready queue), and there are pointers from each process structure to the process' parent, to its youngest living child, and to various other relatives of interest, such as a list of processes sharing the same program code (text).

The *virtual address space* of a user process is divided into text (program code), data, and stack segments. The data and stack segments are always in the same address space, but may grow separately, and usually in opposite directions: most frequently, the stack grows down as the data grow up toward it. The text segment is sometimes (as on an Intel 8086 with separate instruction and data space) in an address space different from the data and stack, and is usually read-only. The debugger puts a text segment in read-write mode to be able to allow insertion of breakpoints.

Every process with sharable text (almost all, under 4.3BSD) has a pointer from its process structure to a *text structure*. The text structure records how many processes are using the text segment, including a pointer into a list of their process structures, and where the page table for the text segment can be found on disk when it is swapped. The text structure itself is always resident in main memory: an array of such structures is allocated at system link time. The text, data, and stack segments for the processes may be swapped. When the segments are swapped in, they are paged.

The *page tables* record information on the mapping from the process' virtual memory to physical memory. The process structure contains pointers to the page table, for use when the process is resident in main memory, or the address of the process on the swap device, when the process is swapped. There is no special separate page table for a shared text segment; every process sharing the text segment has entries for its pages in the process' page table.

Information about the process that is needed only when the process is resident (that is, not swapped out) is kept in the *user structure* (or *u structure*), rather than in the process structure. The *u* structure is mapped read-only into user virtual address space, so user processes can read its contents. It is writable by the kernel. On the VAX, a copy of the VAX PCB is kept here for saving the process' general registers, stack pointer, program counter, and page-table base registers when the process is not running. There is space to keep system-call parameters and return values. All user and group identifiers associated with the process (not just the effective user identifier kept in the process structure) are kept here. Signals, timers, and quotas have data structures here. Of more obvious relevance to the ordinary user, the current directory and the table of open files are maintained in the user structure.

Every process has both a user and a system phase. Most ordinary work is done in *user mode*, but, when a system call is made, it is performed in

system mode. The system and user phases of a process never execute simultaneously. When a process is executing in system mode, a *kernel stack* for that process is used, rather than the user stack belonging to that process. The kernel stack for the process immediately follows the user structure: The kernel stack and the user structure together compose the *system data segment* for the process. The kernel has its own stack for use when it is not doing work on behalf of a process (for instance, for interrupt handling).

Figure 19.6 illustrates how the process structure is used to find the various parts of a process.

The **fork** system call allocates a new process structure (with a new process identifier) for the child process, and copies the user structure. There is ordinarily no need for a new text structure, as the processes share their text; the appropriate counters and lists are merely updated. A new page table is constructed, and new main memory is allocated for the data and stack segments of the child process. The copying of the user structure preserves open file descriptors, user and group identifiers, signal handling, and most similar properties of a process.

The **vfork** system call does *not* copy the data and stack to the new process; rather, the new process simply shares the page table of the old one. A new user structure and a new process structure are still created. A common use of this system call is by a shell to execute a command and to

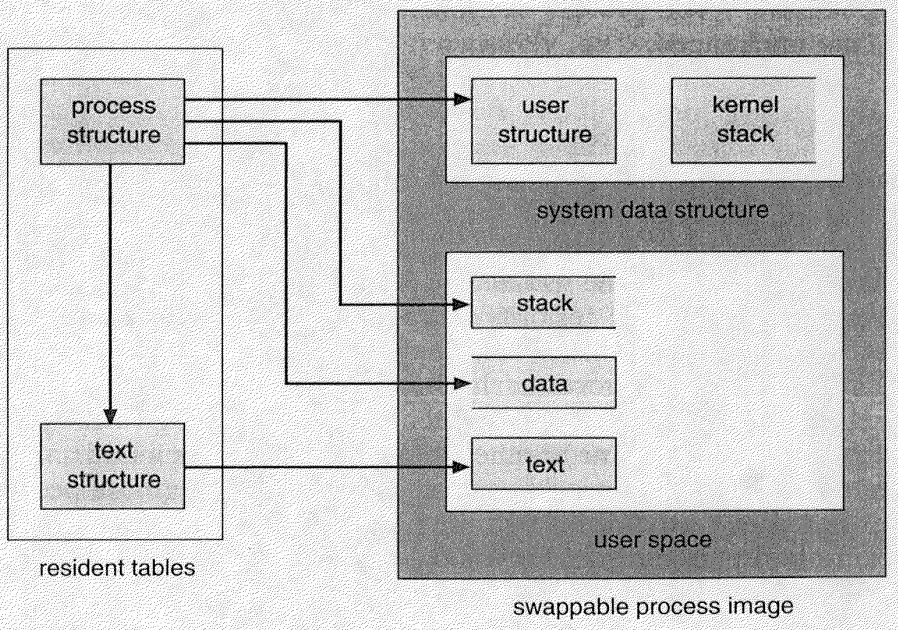

Figure 19.6 Finding parts of a process using the process structure.

wait for its completion. The parent process uses **vfork** to produce the child process. Because the child process wishes to use an **execve** immediately to change its virtual address space completely, there is no need for a complete copy of the parent process. Such data structures as are necessary for manipulating pipes may be kept in registers between the **vfork** and the **execve**. Files may be closed in one process without affecting the other process, since the kernel data structures involved depend on the user structure, which is not shared. The parent is suspended when it calls **vfork** until the child either calls **execve** or terminates, so that the parent will not change memory that the child needs.

When the parent process is large, **vfork** can produce substantial savings in system CPU time. However, it is a fairly dangerous system call, since any memory change occurs in both processes until the **execve** occurs. An alternative is to share all pages by duplicating the page table, but to mark the entries of both page tables as *copy-on-write*. The hardware protection bits are set to trap any attempt to write in these shared pages. If such a trap occurs, a new frame is allocated and the shared page is copied to the new frame. The page tables are adjusted to show that this page is no longer shared (and therefore no longer needs to be write-protected), and execution can resume.

An **execve** system call creates no new process or user structure; rather, the text and data of the process are replaced. Open files are preserved (although there is a way to specify that certain file descriptors are to be closed on an **execve**). Most signal-handling properties are preserved, but arrangements to call a specific user routine on a signal are canceled, for obvious reasons. The process identifier and most other properties of the process are unchanged.

19.5.2 CPU Scheduling

CPU scheduling in UNIX is designed to benefit interactive processes. Processes are given small CPU time slices by a priority algorithm that reduces to round-robin scheduling for CPU-bound jobs.

Every process has a *scheduling priority* associated with it; larger numbers indicate lower priority. Processes doing disk I/O or other important tasks have priorities less than "pzero" and cannot be killed by signals. Ordinary user processes have positive priorities and thus are all less likely to be run than are any system process, although user processes can set precedence over one another through the *nice* command.

The more CPU time a process accumulates, the lower (more positive) its priority becomes, and vice versa, so there is negative feedback in CPU scheduling and it is difficult for a single process to take all the CPU time. Process aging is employed to prevent starvation.

Older UNIX systems used a 1-second quantum for the round-robin scheduling. 4.3BSD reschedules processes every 0.1 second and recomputes

priorities every second. The round-robin scheduling is accomplished by the *timeout* mechanism, which tells the clock interrupt driver to call a kernel subroutine after a specified interval; the subroutine to be called in this case causes the rescheduling and then resubmits a *timeout* to call itself again. The priority recomputation is also timed by a subroutine that resubmits a *timeout* for itself.

There is no preemption of one process by another in the kernel. A process may relinquish the CPU because it is waiting on I/O or because its time slice has expired. When a process chooses to relinquish the CPU, it goes to sleep on an *event*. The kernel primitive used for this purpose is called *sleep* (not to be confused with the user-level library routine of the same name). It takes an argument, which is by convention the address of a kernel data structure related to an *event* that the process wants to occur before that process is awakened. When the event occurs, the system process that knows about it calls *wakeup* with the address corresponding to the event, and *all* processes that had done a *sleep* on the same address are put in the ready queue to be run.

For example, a process waiting for disk I/O to complete will *sleep* on the address of the buffer header corresponding to the data being transferred. When the interrupt routine for the disk driver notes that the transfer is complete, it calls *wakeup* on the buffer header. The interrupt uses the kernel stack for whatever process happened to be running at the time, and the *wakeup* is done from that system process.

The process that actually does run is chosen by the scheduler. *Sleep* takes a second argument, which is the scheduling priority to be used for this purpose. This priority argument, if less than "pzero", also prevents the process from being awakened prematurely by some exceptional event, such as a *signal*.

When a signal is generated, it is left pending until the system half of the affected process next runs. This event usually happens soon, since the signal normally causes the process to be awakened if the process has been waiting for some other condition.

There is no memory associated with events, and the caller of the routine that does a *sleep* on an event must be prepared to deal with a premature return, including the possibility that the reason for waiting has vanished.

There are *race conditions* involved in the event mechanism. If a process decides (because of checking a flag in memory, for instance) to sleep on an event, and the event occurs before the process can execute the primitive that does the actual sleep on the event, the process sleeping may then sleep forever. We prevent this situation by raising the hardware processor priority during the critical section so that no interrupts can occur, and thus only the process desiring the event can run until it is sleeping. Hardware processor priority is used in this manner to protect critical regions throughout the kernel, and is the greatest obstacle to porting UNIX to

multiple processor machines. However, this problem has not prevented such ports from being done repeatedly.

Many processes such as text editors are I/O bound and usually will be scheduled mainly on the basis of waiting for I/O. Experience suggests that the UNIX scheduler performs best with I/O-bound jobs, as can be observed when there are several CPU-bound jobs, such as text formatters or language interpreters, running.

What has been referred to here as *CPU scheduling* corresponds closely to the *short-term scheduling* of Chapter 4, although the negative-feedback property of the priority scheme provides some long-term scheduling in that it largely determines the long-term *job mix*. Medium-term scheduling is done by the swapping mechanism described in Section 19.6.

19.6 ■ Memory Management

Much of UNIX's early development was done on a PDP-11. The PDP-11 has only eight segments in its virtual address space, and each of these are at most 8192 bytes. The larger machines, such as the PDP-11/70, allow separate instruction and address spaces, which effectively double the address space and number of segments, but this address space is still relatively small. In addition, the kernel was even more severely constrained due to dedication of one data segment to interrupt vectors, another to point at the per-process system data segment, and yet another for the UNIBUS (system I/O bus) registers. Further, on the smaller PDP-11s, total physical memory was limited to 256K. The total memory resources were insufficient to justify or support complex memory-management algorithms. Thus, UNIX swapped entire process memory images.

19.6.1 Swapping

Pre-3BSD UNIX systems use swapping exclusively to handle memory contention among processes: If there is too much contention, processes are swapped out until enough memory is available. Also, a few large processes can force many small processes out of memory, and a process larger than nonkernel main memory cannot be run at all. The system data segment (the *u* structure and kernel stack) and the user data segment (text [if nonsharable], data, and stack) are kept in contiguous main memory for swap-transfer efficiency, so external fragmentation of memory can be a serious problem.

Allocation of both main memory and swap space is done first-fit. When the size of a process' memory image increases (due to either stack expansion or data expansion), a new piece of memory big enough for the whole image is allocated. The memory image is copied, the old memory is

freed, and the appropriate tables are updated. (An attempt is made in some systems to find memory contiguous to the end of the current piece, to avoid some copying.) If no single piece of main memory is large enough, the process is swapped out such that it will be swapped back in with the new size.

There is no need to swap out a sharable text segment, because it is read-only, and there is no need to read in a sharable text segment for a process when another instance is already in core. That is one of the main reasons for keeping track of sharable text segments: less swap traffic. The other reason is the reduced amount of main memory required for multiple processes using the same text segment.

Decisions regarding which processes to swap in or out are made by the *scheduler process* (also known as the *swapper*). The *scheduler* wakes up at least once every 4 seconds to check for processes to be swapped in or out. A process is more likely to be swapped out if it is idle, has been in main memory a long time, or is large; if no obvious candidates are found, other processes are picked by age. A process is more likely to be swapped in if it has been swapped out a long time, or is small. There are checks to prevent thrashing, basically by not letting a process be swapped out if it has not been in memory for a certain amount of time.

If jobs do not need to be swapped out, the process table is searched for a process deserving to be brought in (determined by how small the process is and how long it has been swapped out). If there is not enough memory available, processes are swapped out until there is.

In 4.3BSD, swap space is allocated in pieces that are multiples of a power of 2 and a minimum size (for example, 32 pages), up to a maximum that is determined by the size of the swap-space partition on the disk. If several logical disk partitions may be used for swapping, they should be the same size, for this reason. The several logical disk partitions should also be on separate disk arms to minimize disk seeks.

Many UNIX systems still use the swapping scheme just described. All Berkeley UNIX systems, on the other hand, depend primarily on paging for memory-contention management, and depend only secondarily on swapping. A scheme similar in outline to the traditional one is used to determine which processes get swapped in or out, but the details differ and the influence of swapping is less.

19.6.2 Paging

Berkeley introduced paging to UNIX with 3BSD. VAX 4.2BSD is a demand-paged virtual-memory system. External fragmentation of memory is eliminated by paging. (There is, of course, internal fragmentation, but it is negligible with a reasonably small page size.) Swapping can be kept to a minimum because more jobs can be kept in main memory, because paging allows execution with only parts of each process in memory.

Demand paging is done in a straightforward manner. When a process needs a page and the page is not there, a page fault to the kernel occurs, a frame of main memory is allocated, and the proper disk page is read into the frame.

There are a few optimizations. If the page needed is still in the page table for the process, but has been marked invalid by the page-replacement process, it can be marked valid and used without any I/O transfer. Pages can similarly be retrieved from the list of free frames. When most processes are started, many of their pages are prepaged and are put on the free list for recovery by this mechanism. Arrangements may also be made for a process to have no prepaging on startup, but that is seldom done, as it results in more page-fault overhead, being closer to pure demand paging.

If the page has to be fetched from disk, it must be locked in memory for the duration of the transfer. This locking ensures that the page will not be selected for page replacement. Once the page is fetched and mapped properly, it must remain locked if raw physical I/O is being done on it.

The *page-replacement* algorithm is more interesting. The VAX has no hardware memory page-reference bit. This lack makes many memory-management algorithms, such as page-fault frequency, unusable. 4.2BSD uses a modification of the *second chance* (clock) algorithm described in Section 9.5.4. The map of all nonkernel main memory (the *core map* or *cmap*) is swept linearly and repeatedly by a software *clock hand*. When the clock hand reaches a given frame, if the frame is marked as in use by some software condition (for example, physical I/O is in progress using it), or the frame is already free, the frame is left untouched, and the clock hand sweeps to the next frame. Otherwise, the corresponding text or process page-table entry for this frame is located. If the entry is already invalid, the frame is added to the free list; otherwise, the page-table entry is made invalid but reclaimable (that is, if it does not get paged out by the next time it is wanted, it can just be made valid again). 4.3BSD Tahoe added support for systems which do implement the reference bit. On such systems, one pass of the clock turns the reference bit off, and a second pass places those pages whose reference bits remain off onto the free list for replacement. Of course, if the page is dirty (the VAX *does* have a dirty bit), it must first be written to disk before being added to the free list. Pageouts are done in clusters to improve performance.

There are checks to make sure that the number of valid data pages for a process does not fall too low, and to keep the paging device from being flooded with requests. There is also a mechanism by which a process may limit the amount of main memory it uses.

The LRU clock hand is implemented in the *pagedaemon*, which is process 2 (remember that the *scheduler* is process 0, and *init* is process 1). This process spends most of its time sleeping, but a check is done several times per second (scheduled by a *timeout*) to see if action is necessary; if it is, process 2 is awakened. Whenever the number of free frames falls below a

threshold, *lotsfree*, the *pagedaemon* is awakened; thus, if there is always a large amount of free memory, the *pagedaemon* imposes no load on the system, because it never runs.

The sweep of the clock hand each time the *pagedaemon* process is awakened (that is, the number of frames scanned, which is usually more than the number paged out), is determined both by the number of frames lacking to reach *lotsfree* and by the number of frames that the *scheduler* has determined are needed for various reasons (the more frames needed, the longer the sweep). If the number of frames free rises to *lotsfree* before the expected sweep is completed, the hand stops and the *pagedaemon* process sleeps. The parameters that determine the range of the clock-hand sweep are determined at system startup according to the amount of main memory, such that *pagedaemon* does not use more than 10 percent of all CPU time.

If the *scheduler* decides that the paging system is overloaded, processes will be swapped out whole until the overload is relieved. This swapping usually happens only if several conditions are met: load average is high, free memory has fallen below a low limit, *minfree;* and the average memory available over recent time is less than a desirable amount, *desfree*, where *lotsfree* > *desfree* > *minfree*. In other words, only a chronic shortage of memory with several processes trying to run will cause swapping, and even then free memory has to be extremely low at the moment. (An excessive paging rate or a need for memory by the kernel itself may also enter into the calculations, in rare cases.) Processes may be swapped by the *scheduler*, of course, for other reasons (such as simply for not running for a long time).

The parameter *lotsfree* is usually one-quarter of the memory in the map that the clock hand sweeps, and *desfree* and *minfree* are usually the same across different systems, but are limited to fractions of available memory.

Every process' text segment is by default shared and read-only. This scheme is practical with paging, because there is no external fragmentation, and the swap space gained by sharing more than offsets the negligible amount of overhead involved, as the kernel virtual space is large.

CPU scheduling, swapping, and paging interact: the lower the priority of a process, the more likely that its pages will be paged out and the more likely that it will be swapped in its entirety.

The age preferences in choosing processes to swap guard against thrashing, but paging does so more effectively. Ideally, processes will not be swapped out unless they are idle, because each process will need only a small working set of pages in main memory at any one time, and the *pagedaemon* will reclaim unused pages for use by other processes.

The amount of memory the process will need is some fraction of that process' total virtual size, up to one-half if that process has been swapped out for a long time.

The VAX 512-byte hardware pages are too small for I/O efficiency, so they are clustered in groups of two so that all paging I/O is actually done in 1024-byte chunks. In other words, the effective page size is not necessarily identical to the hardware page size of the machine, although it must be a multiple of the hardware page size.

19.7 ■ File System

The UNIX file system supports two main objects: files and directories. Directories are just files with a special format, so the representation of a file is the basic UNIX concept.

19.7.1 Blocks and Fragments

Most of the file system is taken up by *data blocks*, which contain whatever the users have put in their files. Let us consider how these data blocks are stored on the disk.

The hardware disk sector is usually 512 bytes. A block size larger than 512 bytes is desirable for speed. However, because UNIX file systems usually contain a very large number of small files, much larger blocks would cause excessive internal fragmentation. That is why the earlier 4.1BSD file system was limited to a 1024-byte (1K) block.

The 4.2BSD solution is to use *two* block sizes for files which have no indirect blocks: all the blocks of a file are of a large *block* size (such as 8K), except the last. The last block is an appropriate multiple of a smaller *fragment* size (for example, 1024) to fill out the file. Thus, a file of size 18,000 bytes would have two 8K blocks and one 2K fragment (which would not be filled completely).

The *block* and *fragment* sizes are set during file-system creation according to the intended use of the file system: If many small files are expected, the fragment size should be small; if repeated transfers of large files are expected, the basic block size should be large. Implementation details force a maximum block-to-fragment ratio of 8 : 1, and a minimum block size of 4K, so typical choices are 4096 : 512 for the former case and 8192 : 1024 for the latter.

Suppose data are written to a file in transfer sizes of 1K bytes, and the block and fragment sizes of the file system are 4K and 512 bytes. The file system will allocate a 1K fragment to contain the data from the first transfer. The next transfer will cause a new 2K fragment to be allocated. The data from the original fragment must be copied into this new fragment, followed by the second 1K transfer. The allocation routines do attempt to find the required space on the disk immediately following the

existing fragment so that no copying is necessary, but, if they cannot do so, up to seven copies may be required before the fragment becomes a block. Provisions have been made for programs to discover the block size for a file so that transfers of that size can be made, to avoid fragment recopying.

19.7.2 Inodes

A file is represented by an *inode* (Figure 11.7). An inode is a record that stores most of the information about a specific file on the disk. The name *inode* (pronounced *EYE node*) is derived from "index node" and was originally spelled "i-node"; the hyphen fell out of use over the years. The term is also sometimes spelled "I node".

The inode contains the user and group identifiers of the file, the times of the last file modification and access, a count of the number of hard links (directory entries) to the file, and the type of the file (plain file, directory, symbolic link, character device, block device, or socket). In addition, the inode contains 15 pointers to the disk blocks containing the data contents of the file. The first 12 of these pointers point to *direct blocks;* that is, they contain addresses of blocks that contain data of the file. Thus, the data for small files (no more than 12 blocks) can be referenced immediately, because a copy of the inode is kept in main memory while a file is open. If the block size is 4K, then up to 48K of data may be accessed directly from the inode.

The next three pointers in the inode point to *indirect blocks*. If the file is large enough to use indirect blocks, the indirect blocks are each of the major block size; the fragment size applies to only data blocks. The first indirect block pointer is the address of a *single indirect block*. The single indirect block is an index block, containing not data, but rather the addresses of blocks that do contain data. Then, there is a *double-indirect-block pointer,* the address of a block that contains the addresses of blocks that contain pointers to the actual data blocks. The last pointer would contain the address of a *triple indirect block;* however, there is no need for it. The minimum block size for a file system in 4.2BSD is 4K, so files with as many as 2^{32} bytes will use only double, not triple, indirection. That is, as each block pointer takes 4 bytes, we have 49,152 bytes accessible in direct blocks, 4,194,304 bytes accessible by a single indirection, and 4,294,967,296 bytes reachable through double indirection, for a total of 4,299,210,752 bytes, which is larger than 2^{32} bytes. The number 2^{32} is significant because the file offset in the file structure in main memory is kept in a 32-bit word. Files therefore cannot be larger than 2^{32} bytes. Since file pointers are signed integers (for seeking backward and forward in a file), the actual maximum file size is 2^{32-1} bytes. Two gigabytes is large enough for most purposes.

19.7.3 Directories

There is no distinction between plain files and directories at this level of implementation; directory contents are kept in data blocks, and directories are represented by an inode in the same way as plain files. Only the inode type field distinguishes between plain files and directories. Plain files are not assumed to have a structure, however, whereas directories have a specific structure. In Version 7, file names were limited to 14 characters, so directories were a list of 16-byte entries: 2 bytes for an inode number and 14 bytes for a file name.

In 4.2BSD, file names are of variable length, up to 255 bytes, so directory entries are also of variable length. Each entry contains first the length of the entry, then the file name and the inode number. This variable-length entry makes the directory management and search routines more complex, but greatly improves the ability of users to choose meaningful names for their files and directories, with no practical limit on the length of the name.

The first two names in every directory are "." and "..". New directory entries are added to the directory in the first space available, generally after the existing files. A linear search is used.

The user refers to a file by a path name, whereas the file system uses the inode as its definition of a file. Thus, the kernel has to map the supplied user path name to an inode. The directories are used for this mapping.

First, a starting directory is determined. If the first character of the path name is "/", the starting directory is the root directory. If the path name starts with any character other than a slash, the starting directory is the current directory of the current process. The starting directory is checked for proper file type and access permissions, and an error is returned if necessary. The inode of the starting directory is always available.

The next element of the path name, up to the next "/", or to the end of the path name, is a file name. The starting directory is searched for this name, and an error is returned if the name is not found. If there is yet another element in the path name, the current inode must refer to a directory, and an error is returned if it does not, or if access is denied. This directory is searched as was the previous one. This process continues until the end of the path name is reached and the desired inode is returned. This step-by-step process is needed because at any directory a mount point (or symbolic link, see below) may be encountered, causing the translation to move to a different directory structure for continuation.

Hard links are simply directory entries like any other. We handle symbolic links for the most part by starting the search over with the path name taken from the contents of the symbolic link. We prevent infinite

loops by counting the number of symbolic links encountered during a path-name search and returning an error when a limit (eight) is exceeded.

Nondisk files (such as devices) do not have data blocks allocated on the disk. The kernel notices these file types (as indicated in the inode) and calls appropriate drivers to handle I/O for them.

Once the inode is found by, for instance, the **open** system call, a *file structure* is allocated to point to the inode. The file descriptor given to the user refers to this file structure. 4.3BSD added a *directory name cache* to hold recent directory-to-inode translations. This improvement greatly increased file system performance.

19.7.4 Mapping of a File Descriptor to an Inode

System calls that refer to open files indicate the file by passing a file descriptor as an argument. The file descriptor is used by the kernel to index a table of open files for the current process. Each entry of the table contains a pointer to a file structure. This file structure in turn points to the inode; see Figure 19.7. The open file table has a fixed length which is only settable at boot time. Therefore, there is a fixed limit on the number of concurrently open files in a system.

The **read** and **write** system calls do not take a position in the file as an argument. Rather, the kernel keeps a *file offset*, which is updated by an appropriate amount after each **read** or **write** according to the number of data actually transferred. The offset can be set directly by the **lseek** system call. If the file descriptor indexed an array of inode pointers instead of file pointers, this offset would have to be kept in the inode. Because more than one process may open the same file, and each such process needs its

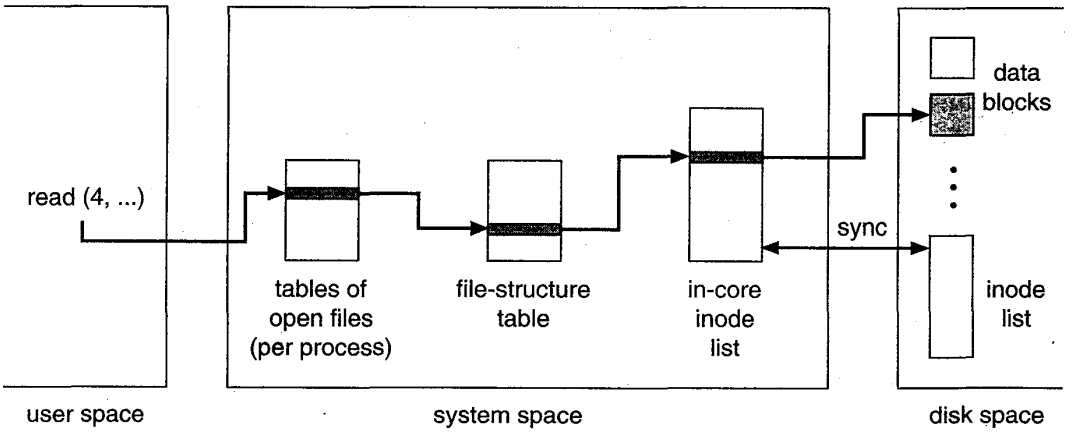

Figure 19.7 File-system control blocks.

own offset for the file, keeping the offset in the inode is inappropriate. Thus, the file structure is used to contain the offset.

File structures are inherited by the child process after a **fork**, so several processes may share the *same* offset location for a file.

The *inode structure* pointed to by the file structure is an in-core copy of the inode on the disk. The in-core inode has a few extra fields, such as a reference count of how many file structures are pointing at it, and the file structure has a similar reference count for how many file descriptors refer to it. When a count becomes zero, the entry is no longer needed and may be reclaimed and reused.

19.7.5 Disk Structures

The file system that the user sees is supported by data on a mass storage device — usually, a disk. The user ordinarily knows of only one file system, but this one logical file system may actually consist of several *physical* file systems, each on a different device. Because device characteristics differ, each separate hardware device defines its own physical file system. In fact, it is generally desirable to partition large physical devices, such as disks, into multiple *logical* devices. Each logical device defines a physical file system. Figure 19.8 illustrates how a directory structure is partitioned into file systems, which are mapped onto logical devices, which are partitions of physical devices. The sizes and locations of these partitions were coded into device drivers in earlier systems, but are maintained on the disk by 4.3BSD.

Partitioning a physical device into multiple file systems has several benefits. Different file systems can support different uses. Although most partitions would be used by the file system, at least one will be necessary for a swap area for the virtual-memory software. Reliability is improved, because software damage is generally limited to only one file system. We can improve efficiency by varying the file-system parameters (such as the block and fragment sizes) for each partition. Also, separate file systems prevent one program from using all available space for a large file, because files cannot be split across file systems. Finally, disk backups are done per partition, and it is faster to search a backup tape for a file if the partition is smaller. Restoring the full partition from tape is also faster.

The actual number of file systems on a drive varies according to the size of the disk and the purpose of the computer system as a whole. One file system, the *root file system*, is always available. Other file systems may be *mounted* — that is, integrated into the directory hierarchy of the root file system.

A bit in the inode structure indicates that the inode has a file system mounted on it. A reference to this file causes the *mount table* to be searched to find the device number of the mounted device. The device number is

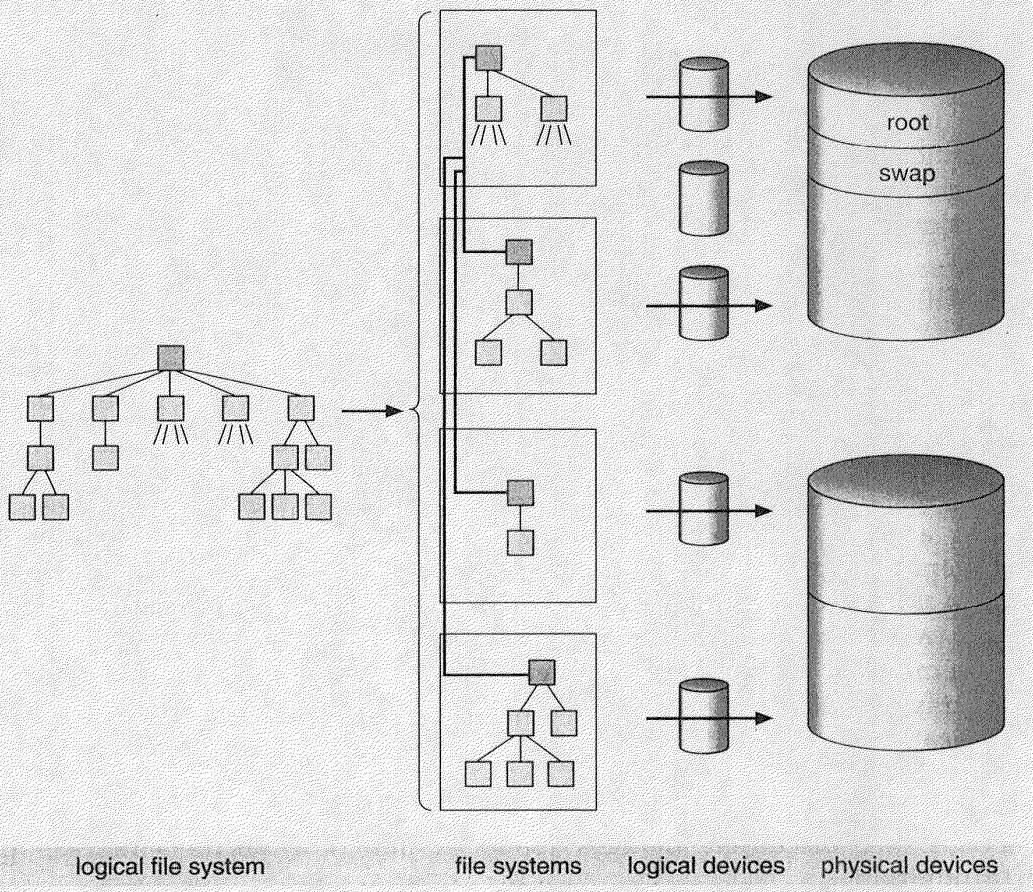

Figure 19.8 Mapping of a logical file system to physical devices.

used to find the inode of the root directory of the mounted file system, and that inode is used. Conversely, if a path-name element is ".." and the directory being searched is the root directory of a file system that is mounted, the mount table is searched to find the inode it is mounted on, and that inode is used.

Each file system is a separate system resource and represents a set of files. The first sector on the logical device is the *boot block,* possibly containing a primary bootstrap program, which may be used to call a secondary bootstrap program residing in the next 7.5K. A system needs only one partition containing boot-block data, but duplicates may be installed via privileged programs by the systems manager, to allow booting when the primary copy is damaged. The *superblock* contains static parameters of the file system. These parameters include the total size of the file system, the block and fragment sizes of the data blocks, and assorted parameters that affect allocation policies.

19.7.6 Implementations

The user interface to the file system is simple and well defined, allowing the implementation of the file system itself to be changed without significant effect on the user. The file system was changed between Version 6 and Version 7, and again between Version 7 and 4BSD. For Version 7, the size of inodes doubled, the maximum file and file-system sizes increased, and the details of free-list handling and superblock information changed. At that time also, *seek* (with a 16-bit offset) became *lseek* (with a 32-bit offset), to allow specification of offsets in larger files, but few other changes were visible outside the kernel.

In 4.0BSD, the size of blocks used in the file system was increased from 512 bytes to 1024 bytes. Although this increased size produced increased internal fragmentation on the disk, it doubled throughput, due mainly to the greater number of data accessed on each disk transfer. This idea was later adopted by System V, along with a number of other ideas, device drivers, and programs.

4.2BSD added the Berkeley Fast File System, which increased speed, and was accompanied by new features. Symbolic links required new system calls. Long file names necessitated the new directory system calls to traverse the now-complex internal directory structure. Finally, the **truncate** calls were added. The Fast File System was a success, and is now found in most implementations of UNIX. Its performance is made possible by its layout and allocation policies, which we discuss next. In Section 11.2.4, we discussed changes made in SunOS to further increase disk throughput.

19.7.7 Layout and Allocation Policies

The kernel uses a *<logical device number, inode number>* pair to identify a file. The logical device number defines the file system involved. The inodes in the file system are numbered in sequence. In the Version 7 file system, all inodes are in an array immediately following a single superblock at the beginning of the logical device, with the data blocks following the inodes. The *inode number* is effectively just an index into this array.

With the Version 7 file system, a block of a file can be anywhere on the disk between the end of the inode array and the end of the file system. Free blocks are kept in a linked list in the superblock. Blocks are pushed onto the front of the free list, and are removed from the front as needed to serve new files or to extend existing files. Thus, the blocks of a file may be arbitrarily far from both the inode and one another. Furthermore, the more a file system of this kind is used, the more disorganized the blocks in a file become. We can reverse this process only by reinitializing and restoring the entire file system, which is not a convenient task to perform. This process was described in Section 11.6.2.

Another difficulty is that the reliability of the file system is suspect. For speed, the superblock of each mounted file system is kept in memory. Keeping the superblock in memory allows the kernel to access a superblock quickly, especially for using the free list. Every 30 seconds, the superblock is written to the disk, to keep the in-core and disk copies synchronized (by the update program, using the **sync** system call). However, it is not uncommon for system bugs or hardware failures to cause a system crash, which destroys the in-core superblock between updates to the disk. Then, the free list on disk does not reflect accurately the state of the disk; to reconstruct it, we must perform a lengthy examination of all blocks in the file system. Note that this problem still remains in the new file system.

The 4.2BSD file-system implementation is radically different from that of Version 7. This reimplementation was done primarily to improve efficiency and robustness, and most such changes are invisible outside the kernel. There were other changes introduced at the same time, such as symbolic links and long file names (up to 255 characters), that are visible at both the system-call and the user levels. Most of the changes required for these features were not in the kernel, however, but rather were in the programs that use them.

Space allocation is especially different. The major new concept in 4.3BSD is the *cylinder group*. The cylinder group was introduced to allow localization of the blocks in a file. Each cylinder group occupies one or more consecutive cylinders of the disk, so that disk accesses within the cylinder group require minimal disk head movement. Every cylinder group has a superblock, a cylinder block, an array of inodes, and some data blocks (Figure 19.9).

The superblock is identical in each cylinder group, so that it can be recovered from any one of them in the event of disk corruption. The *cylinder block* contains dynamic parameters of the particular cylinder group. These include a bit map of free data blocks and fragments, and a bit map

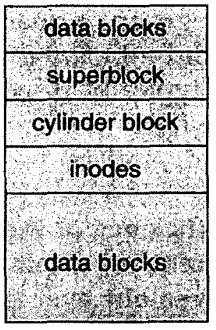

Figure 19.9 4.3BSD cylinder group.

of free inodes. Statistics on recent progress of the allocation strategies are also kept here.

The header information in a cylinder group (the superblock, the cylinder block, and the inodes) is not always at the beginning of the cylinder group. If it were, the header information for every cylinder group might be on the same disk platter; a single disk head crash could wipe out all of them. Therefore, each cylinder group has its header information at a different offset from the beginning of the group.

It is common for the directory-listing command *ls* to read all the inodes of every file in a directory, making it desirable for all such inodes to be close together. For this reason, the inode for a file is usually allocated from the same cylinder group as is the inode of the file's parent directory. Not everything can be localized, however, so an inode for a new directory is put in a *different* cylinder group from that of its parent directory. The cylinder group chosen for such a new directory inode is that with the greatest number of unused inodes.

To reduce disk head seeks involved in accessing the data blocks of a file, we allocate blocks from the same cylinder group as often as possible. Because a single file cannot be allowed to take up all the blocks in a cylinder group, a file exceeding a certain size (such as 2 megabytes) has further block allocation redirected to a different cylinder group, the new group being chosen from among those having more than average free space. If the file continues to grow, allocation is again redirected (at each megabyte) to yet another cylinder group. Thus, all the blocks of a small file are likely to be in the same cylinder group, and the number of long head seeks involved in accessing a large file is kept small.

There are two levels of disk-block-allocation routines. The global policy routines select a desired disk block according to the considerations already discussed. The local policy routines use the specific information recorded in the cylinder blocks to choose a block near the one requested. If the requested block is not in use, it is returned. Otherwise, the block rotationally closest to the one requested in the same cylinder, or a block in a different cylinder but in the same cylinder group, is returned. If there are no more blocks in the cylinder group, a quadratic rehash is done among all the other cylinder groups to find a block; if that fails, an exhaustive search is done. If enough free space (typically 10 percent) is left in the file system, blocks usually are found where desired, the quadratic rehash and exhaustive search are not used, and performance of the file system does not degrade with use.

Because of the increased efficiency of the Fast File System, typical disks are now utilized at 30 percent of their raw transfer capacity. This percentage is a marked improvement over that realized with the Version 7 file system, which used about 3 percent of the bandwidth.

4.3BSD Tahoe introduced the Fat Fast File System, which allows the number of inodes per cylinder group, the number of cylinders per cylinder

group, and the number of distinguished rotational positions to be set when the file system is created. 4.3BSD used to set these parameters according to the disk hardware type.

19.8 ■ I/O System

One of the purposes of an operating system is to hide the peculiarities of specific hardware devices from the user. For example, the file system presents a simple consistent storage facility (the file) independent of the underlying disk hardware. In UNIX, the peculiarities of I/O devices are also hidden from the bulk of the kernel itself by the *I/O system*. The I/O system consists of a buffer caching system, general device driver code, and drivers for specific hardware devices. Only the device driver knows the peculiarities of a specific device. The major parts of the I/O system are diagrammed in Figure 19.10.

There are three main kinds of I/O in 4.3BSD: block devices, character devices, and the *socket* interface. The socket interface, together with its protocols and network interfaces, will be treated in Section 19.9.1.

Block devices include disks and tapes. Their distinguishing characteristic is that they are directly addressable in a fixed block size — usually, 512 bytes. A block-device driver is required to isolate details of tracks, cylinders, and so on, from the rest of the kernel. Block devices are accessible directly through appropriate device special files (such as */dev/rp0*), but are more commonly accessed indirectly through the file system. In either case, transfers are buffered through the *block buffer cache*, which has a profound effect on efficiency.

Character devices include terminals and line printers, but also almost everything else (except network interfaces) that does not use the block buffer cache. For instance, */dev/mem* is an interface to physical main memory, and */dev/null* is a bottomless sink for data and an endless source

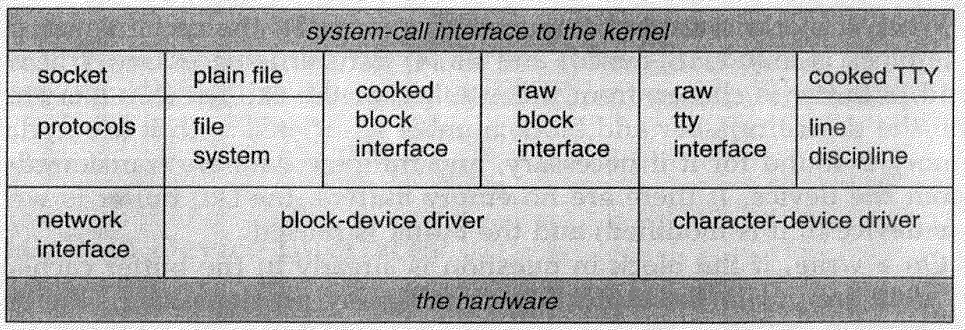

Figure 19.10 4.3BSD kernel I/O structure.

of end-of-file markers. Some devices, such as high-speed graphics interfaces, may have their own buffers or may always do I/O directly into the user's data space; because they do not use the block buffer cache, they are classed as character devices.

Terminals and terminal-like devices use *C-lists*, which are buffers smaller than those of the block buffer cache.

Block devices and *character* devices are the two main device classes. Device drivers are accessed by one of two arrays of entry points. One array is for block devices; the other is for character devices. A device is distinguished by a class (block or character) and a *device number*. The device number consists of two parts. The *major device number* is used to index the array for character or block devices to find entries into the appropriate device driver. The *minor device number* is interpreted by the device driver as, for example, a logical disk partition or a terminal line.

A device driver is connected to the rest of the kernel only by the entry points recorded in the array for its class, and by its use of common buffering systems. This segregation is important for portability, and also for system configuration.

19.8.1 Block Buffer Cache

The block devices use a block buffer cache. The buffer cache consists of a number of buffer headers, each of which can point to a piece of physical memory, as well as to a device number and a block number on the device. The buffer headers for blocks not currently in use are kept in several linked lists, one each for

- Buffers recently used, linked in LRU order (the LRU list)
- Buffers not recently used, or without valid contents (the AGE list)
- EMPTY buffers with no physical memory associated with them

The buffers in these lists are also hashed by device and block number for search efficiency.

When a block is wanted from a device (a read), the cache is searched. If the block is found, it is used, and no I/O transfer is necessary. If it is not found, a buffer is chosen from the AGE list, or the LRU list if AGE is empty. Then the device number and block number associated with it are updated, memory is found for it if necessary, and the new data are transferred into it from the device. If there are no empty buffers, the LRU buffer is written to its device (if it is modified) and the buffer is reused.

On a write, if the block in question is already in the buffer cache, the new data are put in the buffer (overwriting any previous data), the buffer header is marked to indicate the buffer has been modified, and no I/O is immediately necessary. The data will be written when the buffer is needed

for other data. If the block is not found in the buffer cache, an empty buffer is chosen (as with a read) and a transfer is done to this buffer.

Writes are periodically forced for dirty buffer blocks to minimize potential file-system inconsistencies after a crash.

The number of data in a buffer in 4.3BSD is variable, up to a maximum over all file systems, usually 8K. The minimum size is the paging-cluster size, usually 1024 bytes. Buffers are page-cluster aligned, and any page cluster may be mapped into only one buffer at a time, just as any disk block may be mapped into only one buffer at a time. The EMPTY list holds buffer headers which are used if a physical memory block of 8K is split to hold multiple, smaller blocks. Headers are needed for these blocks and are retrieved from EMPTY.

The number of data in a buffer may grow as a user process writes more data following those already in the buffer. When this increase in the data occurs, a new buffer large enough to hold all the data is allocated, and the original data are copied into it, followed by the new data. If a buffer shrinks, a buffer is taken off the empty queue, excess pages are put in it, and that buffer is released to be written to disk.

Some devices, such as magnetic tapes, require blocks to be written in a certain order, so facilities are provided to force a synchronous write of buffers to these devices, in the correct order. Directory blocks are also written synchronously, to forestall crash inconsistencies. Consider the chaos that could occur if many changes were made to a directory, but the directory entries themselves were not updated.

The size of the buffer cache can have a profound effect on the performance of a system, because, if it is large enough, the percentage of cache hits can be high and the number of actual I/O transfers low.

There are some interesting interactions among the buffer cache, the file system, and the disk drivers. When data are written to a disk file, they are buffered in the cache, and the disk driver sorts its output queue according to disk address. These two actions allow the disk driver to minimize disk head seeks and to write data at times optimized for disk rotation. Unless synchronous writes are required, a process writing to disk simply writes into the buffer cache, and the system asynchronously writes the data to disk when convenient. The user process sees very fast writes. When data are read from a disk file, the block I/O system does some read-ahead; however, writes are much nearer to asynchronous than are reads. Thus, output to the disk through the file system is often faster than is input for large transfers, counter to intuition.

19.8.2 Raw Device Interfaces

Almost every block device also has a character interface, and these are called *raw device interfaces*. Such an interface differs from the *block interface* in that the block buffer cache is bypassed.

Each disk driver maintains a queue of pending transfers. Each record in the queue specifies whether it is a read or a write, a main memory address for the transfer (usually in 512-byte increments), a device address for the transfer (usually the address of a disk sector), and a transfer size (in sectors). It is simple to map the information from a block buffer to what is required for this queue.

It is almost as simple to map a piece of main memory corresponding to part of a user process' virtual address space. This mapping is what a raw disk interface, for instance, does. Unbuffered transfers directly to or from a user's virtual address space are thus allowed. The size of the transfer is limited by the physical devices, some of which require an even number of bytes.

The kernel accomplishes transfers for swapping and paging simply by putting the appropriate request on the queue for the appropriate device. No special swapping or paging device driver is needed.

The 4.2BSD file-system implementation was actually written and largely tested as a user process that used a raw disk interface, before the code was moved into the kernel. In an interesting about-face, the Mach operating system (Chapter 20) has no file system per se. File systems can be implemented as user-level tasks.

19.8.3 C-Lists

Terminal drivers use a character buffering system. This system involves keeping small blocks of characters (usually 28 bytes) in linked lists. There are routines to enqueue and dequeue characters for such lists. Although all free character buffers are kept in a single free list, most device drivers that use them limit the number of characters that may be queued at one time for any given terminal line.

A **write** system call to a terminal enqueues characters on a list for the device. An initial transfer is started, and interrupts cause dequeuing of characters and further transfers.

Input is similarly interrupt driven. Terminal drivers typically support *two* input queues, however, and conversion from the first (raw queue) to the other (canonical queue) is triggered by the interrupt routine putting an end-of-line character on the raw queue. The process doing a read on the device is then awakened, and its system phase does the conversion; the characters thus put on the canonical queue are then available to be returned to the user process by the read.

It is also possible to have the device driver bypass the canonical queue and return characters directly from the raw queue. This mode of operation is known as *raw mode*. Full-screen editors, and other programs that need to react to every keystroke, use this mode.

19.9 ■ Interprocess Communication

Many tasks can be accomplished in isolated processes, but many others require interprocess communication. Isolated computing systems have long served for many applications, but networking is increasingly important. With the increasing use of personal workstations, resource sharing is becoming more common. Interprocess communication has not traditionally been one of UNIX's strong points.

Most UNIX systems have not permitted *shared memory* because the PDP-11 hardware did not encourage it. System V does support a shared-memory facility, and one was planned for 4.2BSD, but was not implemented due to time constraints. Solaris 2 supports shared memory, as do many other current versions of UNIX. In any case, shared memory presents a problem in a networked environment, because network accesses can never be as fast as memory accesses on the local machine. Although we could, of course, pretend that memory was shared between two separate machines by copying data across a network transparently, the major benefit of shared memory (speed) would be lost.

19.9.1 Sockets

The *pipe* (discussed in Section 19.4.3) is the IPC mechanism most characteristic of UNIX. A pipe permits a reliable unidirectional byte stream between two processes. It is traditionally implemented as an ordinary file, with a few exceptions. It has no name in the file system, being created instead by the **pipe** system call. Its size is fixed, and when a process attempts to write to a full pipe, the process is suspended. Once all data previously written into the pipe have been read out, writing continues at the beginning of the file (pipes are not true circular buffers). One benefit of the small size (usually 4096 bytes) of pipes is that pipe data are seldom actually written to disk; they usually are kept in memory by the normal block buffer cache.

In 4.3BSD, pipes are implemented as a special case of the *socket* mechanism. The socket mechanism provides a general interface not only to facilities such as pipes, which are local to one machine, but also to networking facilities. Even on the same machine, a pipe can be used only by two processes related through use of the **fork** system call. The socket mechanism can be used by unrelated processes.

A socket is an endpoint of communication. A socket in use usually has an *address* bound to it. The nature of the address depends on the *communication domain* of the socket. A characteristic property of a domain is that processes communicating in the same domain use the same *address format*. A single socket can communicate in only one domain.

The three domains currently implemented in 4.3BSD are the UNIX domain (AF_UNIX), the Internet domain (AF_INET), and the XEROX Network Services (NS) domain (AF_NS). The address format of the UNIX domain is ordinary file-system path names, such as */alpha/beta/gamma*. Processes communicating in the Internet domain use DARPA Internet communications protocols (such as TCP/IP) and Internet addresses, which consist of a 32-bit host number and a 32-bit port number (representing a rendezvous point on the host).

There are several *socket types*, which represent classes of services. Each type may or may not be implemented in any communication domain. If a type is implemented in a given domain, it may be implemented by one or more protocols, which may be selected by the user:

- **Stream sockets:** These sockets provide reliable, duplex, sequenced data streams. No data are lost or duplicated in delivery, and there are no record boundaries. This type is supported in the Internet domain by the TCP protocol. In the UNIX domain, pipes are implemented as a pair of communicating stream sockets.
- **Sequenced packet sockets:** These sockets provide data streams like those of stream sockets, except that record boundaries are provided. This type is used in the XEROX AF_NS protocol.
- **Datagram sockets:** These sockets transfer messages of variable size in either direction. There is no guarantee that such messages will arrive in the same order they were sent, or that they will be unduplicated, or that they will arrive at all, but the original message (record) size is preserved in any datagram that does arrive. This type is supported in the Internet domain by the UDP protocol.
- **Reliably delivered message sockets:** These sockets transfer messages that are guaranteed to arrive, and that otherwise are like the messages transferred using datagram sockets. This type is currently unsupported.
- **Raw sockets:** These sockets allow direct access by processes to the protocols that support the other socket types. The protocols accessible include not only the uppermost ones, but also lower-level protocols. For example, in the Internet domain, it is possible to reach TCP, IP beneath that, or an Ethernet protocol beneath that. This capability is useful for developing new protocols.

The socket facility has a set of system calls specific to it. The **socket** system call creates a socket. It takes as arguments specifications of the communication domain, the socket type, and the protocol to be used to support that type. The value returned is a small integer called a *socket*

descriptor, which is in the same name space as file descriptors. The socket descriptor indexes the array of open "files" in the *u* structure in the kernel, and has a file structure allocated for it. The 4.3BSD file structure may point to a *socket* structure instead of to an inode. In this case, certain socket information (such as the socket's type, message count, and the data in its input and output queues) is kept directly in the socket structure.

For another process to address a socket, the socket must have a name. A name is bound to a socket by the **bind** system call, which takes the socket descriptor, a pointer to the name, and the length of the name as a byte string. The contents and length of the byte string depend on the address format. The **connect** system call is used to initiate a connection. The arguments are syntactically the same as those for **bind;** the socket descriptor represents the local socket and the address is that of the foreign socket to which the attempt to connect is made.

Many processes that communicate using the socket IPC follow the *client–server model*. In this model, the *server* process provides a *service* to the *client* process. When the service is available, the server process listens on a well-known address, and the client process uses **connect,** as described previously, to reach the server.

A server process uses **socket** to create a socket and **bind** to bind the well-known address of its service to that socket. Then, it uses the **listen** system call to tell the kernel that it is ready to accept connections from clients, and to specify how many pending connections the kernel should queue until the server can service them. Finally, the server uses the **accept** system call to accept individual connections. Both **listen** and **accept** take as an argument the socket descriptor of the original socket. **Accept** returns a new socket descriptor corresponding to the new connection; the original socket descriptor is still open for further connections. The server usually uses **fork** to produce a new process after the **accept** to service the client while the original server process continues to listen for more connections.

There are also system calls for setting parameters of a connection and for returning the address of the foreign socket after an **accept.**

When a connection for a socket type such as a stream socket is established, the addresses of both endpoints are known and no further addressing information is needed to transfer data. The ordinary **read** and **write** system calls may then be used to transfer data.

The simplest way to terminate a connection and to destroy the associated socket is to use the **close** system call on its socket descriptor. We may also wish to terminate only one direction of communication of a duplex connection; the **shutdown** system call can be used for this purpose.

Some socket types, such as datagram sockets, do not support connections; instead, their sockets exchange datagrams that must be addressed individually. The system calls **sendto** and **recvfrom** are used for such connections. Both take as arguments a socket descriptor, a buffer

pointer and the length, and an address-buffer pointer and length. The address buffer contains the address to send to for **sendto** and is filled in with the address of the datagram just received by **recvfrom**. The number of data actually transferred is returned by both system calls.

The **select** system call can be used to multiplex data transfers on several file descriptors and/or socket descriptors. It can even be used to allow one server process to listen for client connections for many services and to **fork** a process for each connection as the connection is made. The server does a **socket, bind,** and **listen** for each service, and then does a **select** on all the socket descriptors. When **select** indicates activity on a descriptor, the server does an **accept** on it and forks a process on the new descriptor returned by **accept,** leaving the parent process to do a **select** again.

19.9.2 Network Support

Almost all current UNIX systems support the UUCP network facilities, which are mostly used over dial-up telephone lines to support the UUCP mail network and the USENET news network. These are, however, rudimentary networking facilities, as they do not support even remote login, much less remote procedure call or distributed file systems. These facilities are also almost completely implemented as user processes, and are not part of the operating system proper.

4.3BSD supports the DARPA Internet protocols UDP, TCP, IP, and ICMP on a wide range of Ethernet, token-ring, and ARPANET interfaces. The framework in the kernel to support this is intended to facilitate the implementation of further protocols, and all protocols are accessible via the socket interface. The first version of the code was written by Rob Gurwitz of BBN as an add-on package for 4.1BSD.

The International Standards Organization's (ISO) Open System Interconnection (OSI) Reference Model for networking prescribes seven layers of network protocols and strict methods of communication between them. An implementation of a protocol may communicate only with a peer entity speaking the same protocol at the same layer, or with the protocol-protocol interface of a protocol in the layer immediately above or below in the same system. The ISO networking model is implemented in 4.3BSD Reno and 4.4BSD.

The 4.3BSD networking implementation, and to a certain extent the *socket* facility, is more oriented toward the ARPANET Reference Model (ARM). The ARPANET in its original form served as a proof of concept for many networking concepts, such as packet switching and protocol layering. The ARPANET was retired in 1988 because the hardware that supported it was no longer state of the art. Its successors, such as the NSFNET and the Internet, are even larger, and serve as a communications utility for researchers and as a testbed for Internet gateway research. The ARM

predates the ISO model; the ISO model was in large part inspired by the ARPANET research.

Although the ISO model is often interpreted as requiring a limit of one protocol communicating per layer, the ARM allows several protocols in the same layer. There are only four protocol layers in the ARM, plus

- **Process/Applications:** This layer subsumes the application, presentation, and session layers of the ISO model. Such user-level programs as the File Transfer Protocol (FTP) and Telnet (remote login) exist at this level.
- **Host-Host:** This layer corresponds to ISO's transport and the top part of its network layers. Both the Transmission Control Protocol (TCP) and the Internet Protocol (IP) are in this layer, with TCP on top of IP. TCP corresponds to an ISO transport protocol, and IP performs the addressing functions of the ISO network layer.
- **Network Interface:** This layer spans the lower part of the ISO network layer and all of the data-link layer. The protocols involved here depend on the physical network type. The ARPANET uses the IMP-Host protocols, whereas an Ethernet uses Ethernet protocols.
- **Network Hardware:** The ARM is primarily concerned with software, so there is no explicit network hardware layer; however, any actual network will have hardware corresponding to the ISO hardware layer.

The networking framework in 4.3BSD is more generalized than is either the ISO model or the ARM, although it is most closely related to the ARM; see Figure 19.11.

User processes communicate with network protocols (and thus with other processes on other machines) via the *socket* facility, which corresponds to the ISO Session layer, as it is responsible for setting up and controlling communications.

Sockets are supported by *protocols* — possibly by several, layered one on another. A protocol may provide services such as reliable delivery, suppression of duplicate transmissions, flow control, or addressing, depending on the socket type being supported and the services required by any higher protocols.

A protocol may communicate with another protocol or with the network interface that is appropriate for the network hardware. There is little restriction in the general framework on what protocols may communicate with what other protocols, or on how many protocols may be layered on top of one another. The user process may, by means of the raw socket type, directly access any layer of protocol from the uppermost used to support one of the other socket types, such as streams, down to a raw network interface. This capability is used by routing processes and also for new protocol development.

ISO reference model	ARPANET reference model	4.2BSD layers	example layering
application	process applications	user programs and libraries	telnet
presentation			
session		sockets	sock_stream
transport	host–host	protocol	TCP
network			IP
	network interface	network interfaces	Ethernet driver
data link			
hardware	network hardware	network hardware	interlan controller

Figure 19.11 Network reference models and layering.

There tends to be one *network-interface* driver per network controller type. The network interface is responsible for handling characteristics specific to the local network being addressed. This arrangement ensures that the protocols using the interface do not need to be concerned with these characteristics.

The functions of the network interface depend largely on the *network hardware*, which is whatever is necessary for the network to which it is connected. Some networks may support reliable transmission at this level, but most do not. Some networks provide broadcast addressing, but many do not.

The socket facility and the networking framework use a common set of memory buffers, or *mbuf*s. These are intermediate in size between the large buffers used by the block I/O system and the C-lists used by character devices. An *mbuf* is 128 bytes long, 112 bytes of which may be used for data; the rest is used for pointers to link the *mbuf* into queues and for indicators of how much of the data area is actually in use.

Data are ordinarily passed between layers (socket-protocol, protocol-protocol, or protocol-network interface) in *mbuf*s. This ability to pass the buffers containing the data eliminates some data copying, but there is still frequently a need to remove or add protocol headers. It is also convenient and efficient for many purposes to be able to hold data that occupy an area the size of the memory-management page. Thus, it is possible for the data of an *mbuf* to reside not in the *mbuf* itself, but rather elsewhere in memory. There is an *mbuf* page table for this purpose, as well as a pool of pages dedicated to *mbuf* use.

19.10 ■ Summary

The early advantages of UNIX were that this system was written in a high-level language, was distributed in source form, and had provided powerful operating-system primitives on an inexpensive platform. These advantages led to UNIX's popularity at educational, research, and government institutions, and eventually in the commercial world. This popularity first produced many strains of UNIX with variant and improved facilities. Market pressures are currently leading to the consolidation of these versions. One of the most influential versions is 4.3BSD, developed at Berkeley for the VAX, and later ported to many other platforms.

UNIX provides a file system with tree-structured directories. Files are supported by the kernel as unstructured sequences of bytes. Direct access and sequential access are supported through system calls and library routines.

Files are stored as an array of fixed-size data blocks with perhaps a trailing fragment. The data blocks are found by pointers in the inode. Directory entries point to inodes. Disk space is allocated from cylinder groups to minimize head movement and to improve performance.

UNIX is a multiprogrammed system. Processes can easily create new processes with the **fork** system call. Processes can communicate with pipes or, more generally, sockets. They may be grouped into jobs that may be controlled with signals.

Processes are represented by two structures: the process structure and the user structure. CPU scheduling is a priority algorithm with dynamically computed priorities that reduces to round-robin scheduling in the extreme case.

4.3BSD memory management is swapping supported by paging. A *pagedaemon* process uses a modified second-chance page-replacement algorithm to keep enough free frames to support the executing processes.

Page and file I/O uses a block buffer cache to minimize the amount of actual I/O. Terminal devices use a separate character buffering system.

Networking support is one of the most important features in 4.3BSD. The socket concept provides the programming mechanism to access other processes, even across a network. Sockets provide an interface to several sets of protocols.

■ Exercises

19.1 What are the major differences between 4.3BSD UNIX and SYSVR3? Is one system "better" than the other? Explain your answer.

19.2 How were the design goals of UNIX different from those of other operating systems during the early stages of UNIX development?

19.3 Why are there many different versions of UNIX currently available? In what ways is this diversity an advantage to UNIX? In what ways is it a disadvantage?

19.4 What are the advantages and disadvantages of writing an operating system in a high-level language, such as C?

19.5 In what circumstances is the system-call sequence **fork execve** most appropriate? When is **vfork** preferable?

19.6 Does 4.3BSD UNIX give scheduling priority to I/O or CPU-bound processes? For what reason does it differentiate between these categories, and why is one given priority over the other? How does it know which of these categories fits a given process?

19.7 Early UNIX systems used swapping for memory management, whereas 4.3BSD used paging and swapping. Discuss the advantages and disadvantages of the two memory methods.

19.8 Describe the modifications to a file system that the 4.3BSD kernel makes when a process requests the creation of a new file */tmp/foo* and writes to that file sequentially until the file size reaches 20K.

19.9 Directory blocks in 4.3BSD are written synchronously when they are changed. Consider what would happen if they were written asynchronously. Describe the state of the file system if a crash occurred after all the files in a directory were deleted but before the directory entry was updated on disk.

19.10 Describe the process that is needed to recreate the free list after a crash in 4.1BSD.

19.11 What effects on system performance would the following changes to 4.3BSD have? Explain your answers.

a. The merging of the block buffer cache and the process paging space

b. Clustering disk I/O into larger chunks

c. Implementing and using shared memory to pass data between processes, rather than using RPC or sockets

d. Using the ISO seven-layer networking model, rather than the ARM network model

19.12 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.

Bibliographic Notes

The best general description of the distinctive features of UNIX is still that presented by Ritchie and Thompson [1974]. Much of the history of UNIX was given in Ritchie [1979]. A critique of UNIX was offered by Blair et al. [1985]. The two main modern versions of UNIX are 4.3BSD and System V. System V internals were described at length in Bach [1987]. The authoritative treatment of the design and implementation of 4.3BSD is that by Leffler et al. [1989].

Possibly the best book on general programming under UNIX, especially on the use of the shell and facilities such as *yacc* and *sed,* is that by Kernighan and Pike [1984]. Systems programming was covered by Stevens [1992]. Another text of interest is [Bourne 1983]. The programming language of choice under UNIX is C [Kernighan and Ritchie 1988]. C is also the system's implementation language. The Bourne shell was described in Bourne [1978]. The Korn shell was described in Korn [1983].

The set of documentation that comes with UNIX systems is called the *UNIX Programmer's Manual* (UPM) and is traditionally organized in two volumes. Volume 1 contains short entries for every command, system call, and subroutine package in the system, and is also available on-line via the *man* command. Volume 2, *Supplementary Documents* (usually divided into Volumes 2A and 2B for convenience of binding), contains assorted papers relevant to the system and manuals for those commands or packages too complex to describe in one or two pages. Berkeley systems add Volume 2C to contain documents concerning Berkeley-specific features.

The Version 7 file system was described in Thompson [1978], and the 4.2BSD file system was described in McKusick et al. [1984]. A crash-resistant UNIX file system was described by Anyanwu and Marshall [1986]. The basic reference for process-management is Thompson [1978]. The 3BSD memory-management system was described in Babaoglu and Joy [1981], and some 4.3BSD memory-management developments were described in McKusick and Karels [1988]. The I/O system was described in Thompson [1978].

A description of the UNIX operating-system security was given by Grampp and Morris [1984] and by Wood and Kochan [1985].

Two useful papers on communications under 4.2BSD are those by Leffler et al. [1978, 1983], both in UPM Volume 2C.

The *ISO Reference Model* was given in [ISO 1981]. The *ARPANET Reference Model* was set forth in Cerf and Cain [1983]. The Internet and its protocols were described in Comer [1991], Comer and Stevens [1991, 1993]. UNIX network programming was described thoroughly in Stevens [1990]. The general state of networks was given in Quarterman [1990].

There are many useful papers in the two special issues of *The Bell System Technical Journal* on UNIX [BSTJ 1978, BSTJ 1984]. Other papers of

interest have appeared at various USENIX conferences and are available in the proceedings of those conferences, as well as in the USENIX-refereed journal, *Computer Systems*.

Several textbooks describing variants of the UNIX system are those by Holt [1983], discussing the Tunis operating system; Comer [1984, 1987], discussing the Xinu operating system; and Tanenbaum [1987], describing the Minix operating system.

CHAPTER 20

THE MACH SYSTEM

The Mach operating system is designed to incorporate the many recent innovations in operating-system research to produce a fully functional, technically advanced system. Unlike UNIX, which was developed without regard for multiprocessing, Mach incorporates multiprocessing support throughout. Its multiprocessing support is also exceedingly flexible, ranging from shared memory systems to systems with no memory shared between processors. Mach is designed to run on computer systems ranging from one to thousands of processors. In addition, Mach is easily ported to many varied computer architectures. A key goal of Mach is to be a distributed system capable of functioning on heterogeneous hardware.

Although many experimental operating systems are being designed, built, and used, Mach is better able to satisfy the needs of the masses than the others are because it offers full compatibility with UNIX 4.3BSD. As such, it provides a unique opportunity for us to compare two functionally similar, but internally dissimilar, operating systems. The order and contents of the presentation of Mach is different from that of UNIX to reflect the differing emphasis of the two systems. There is no section on the user interface, because that component is similar in 4.3BSD when running the BSD server. As we shall see, Mach provides the ability to layer emulation of other operating systems as well, and they can even run concurrently.

20.1 ■ History

Mach traces its ancestry to the Accent operating system developed at Carnegie Mellon University (CMU). Although Accent pioneered a number

of novel operating system concepts, its utility was limited by its inability to execute UNIX applications and its strong ties to a single hardware architecture that made it difficult to port. Mach's communication system and philosophy are derived from Accent, but many other significant portions of the system (for example, the virtual memory system, task and thread management) were developed from scratch. An important goal of the Mach effort was support for multiprocessors.

Mach's development followed an evolutionary path from BSD UNIX systems. Mach code was initially developed inside the 4.2BSD kernel, with BSD kernel components being replaced by Mach components as the Mach components were completed. The BSD components were updated to 4.3BSD when that became available. By 1986, the virtual memory and communication subsystems were running on the DEC VAX computer family, including multiprocessor versions of the VAX. Versions for the IBM RT/PC and for SUN 3 workstations followed shortly. 1987 saw the completion of the Encore Multimax and Sequent Balance multiprocessor versions, including task and thread support, as well as the first official releases of the system, Release 0 and Release 1.

Through Release 2, Mach provides compatibility with the corresponding BSD systems by including much of BSD's code in the kernel. The new features and capabilities of Mach make the kernels in these releases larger than the corresponding BSD kernels. Mach 3 (Figure 20.1) moves the BSD code outside of the kernel, leaving a much smaller *microkernel*. This system implements only basic Mach features in the kernel; all UNIX-specific code has been evicted to run in user-mode servers. Excluding UNIX-specific code from the kernel allows replacement of BSD with another operating system, or the simultaneous execution of multiple operating-system interfaces on top of the microkernel. In addition to BSD, user-mode implementations have been developed for DOS, the Macintosh

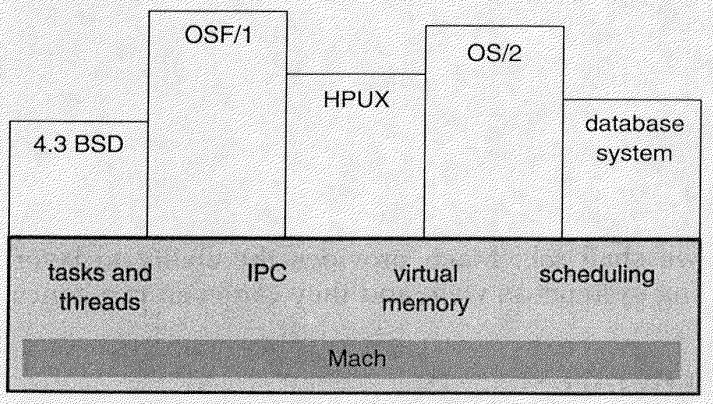

Figure 20.1 Mach 3 structure.

operating system, and OSF/1. This approach has similarities to the *virtual-machine* concept, but the virtual machine is defined by software (the Mach kernel interface), rather than by hardware. As of Release 3.0, Mach became available on a wide variety of systems, including single-processor SUN, Intel, IBM, and DEC machines, and multiprocessor DEC, Sequent, and Encore systems.

Mach was propelled into the forefront of industry attention when the Open Software Foundation (OSF) announced in 1989 that it would use Mach 2.5 as the basis for its new operating system, OSF/1. The initial release of OSF/1 occurred a year later, and now competes with UNIX System V, Release 4, the operating system of choice among *UNIX International (UI)* members. OSF members include key technological companies such as IBM, DEC, and HP. Mach 2.5 is also the basis for the operating system on the *NeXT* workstation, the brainchild of Steve Jobs, of Apple Computer fame. OSF is evaluating Mach 3 as the basis for a future operating-system release, and research on Mach continues at CMU and OSF, and elsewhere.

20.2 ■ Design Principles

The Mach operating system was designed to provide basic mechanisms that most current operating systems lack. The goal is to design an operating system that is BSD compatible and, in addition, excels in the following areas.

- Support for diverse architectures, including multiprocessors with varying degrees of shared memory access: Uniform Memory Access (UMA), Non-Uniform Memory Access (NUMA), and No Remote Memory Access (NORMA)
- Ability to function with varying intercomputer network speeds, from wide-area networks to high-speed local-area networks and tightly coupled multiprocessors
- Simplified kernel structure, with a small number of abstractions; in turn these abstractions are sufficiently general to allow other operating systems to be implemented on top of Mach
- Distributed operation, providing network transparency to clients and an object-oriented organization both internally and externally
- Integrated memory management and interprocess communication, to provide both efficient communication of large numbers of data, and communication-based memory management
- Heterogeneous system support, to make Mach widely available and interoperable among computer systems from multiple vendors

The designers of Mach have been heavily influenced by BSD (and by UNIX in general), whose benefits include

- A simple programmer interface, with a good set of primitives and a consistent set of interfaces to system facilities
- Easy portability to a wide class of uniprocessors
- An extensive library of utilities and applications
- The ability to combine utilities easily via pipes

Of course, BSD was seen as having several drawbacks that need to be redressed:

- A kernel that has become the repository of many redundant features — and that consequently is difficult to manage and modify
- Original design goals that made it difficult to provide support for multiprocessors, distributed systems, and shared program libraries; for instance, because the kernel was designed for uniprocessors, it has no provisions for locking code or data that other processors might be using
- Too many fundamental abstractions, providing too many similar, competing means to accomplish the same task

It should be clear that the development of Mach continues to be a huge undertaking. The benefits of such a system are equally large, however. The operating system runs on many existing uni- and multiprocessor architectures, and can be easily ported to future ones. It makes research easier, because computer scientists can add features via user-level code, instead of having to write their own tailor-made operating system. Areas of experimentation include operating systems, databases, reliable distributed systems, multiprocessor languages, security, and distributed artificial intelligence. In its current instantiation, the Mach system is usually as efficient as are other major versions of UNIX when performing similar tasks.

20.3 ■ System Components

To achieve the design goals of Mach, the developers reduced the operating-system functionality to a small set of basic abstractions, out of which all other functionality can be derived. The Mach approach is to place as little as possible within the kernel, but to make what is there powerful enough that all other features can be implemented at user level.

Mach's design philosophy is to have a simple, extensible kernel, concentrating on communication facilities. For instance, all requests to the kernel, and all data movement among processes, are handled through one communication mechanism. By limiting all data operations to one mechanism, Mach is able to provide systemwide protection to its users by protecting the communications mechanism. Optimizing this one communications path can result in significant performance gains, and is simpler than trying to optimize several paths. Mach is extensible, because many traditionally kernel-based functions can be implemented as user-level servers. For instance, all pagers (including the default pager) can be implemented externally and called by the kernel for the user.

Mach is an example of an *object-oriented* system where the data and the operations that manipulate that data are encapsulated into an abstract object. Only the operations of the object are able to act on the entities defined in it. The details of how these operations are implemented are hidden, as are the internal data structures. Thus, a programmer can use an object only by invoking its defined, exported operations. A programmer can change the internal operations without changing the interface definition, so changes and optimizations do not affect other aspects of system operation. The object-oriented approach supported by Mach allows objects to reside anywhere in a network of Mach systems, transparent to the user. The *port* mechanism, discussed later in this section, makes all of this possible.

Mach's primitive abstractions are the heart of the system, and are as follows:

- A **task** is an execution environment that provides the basic unit of resource allocation. A task consists of a virtual address space and protected access to system resources via ports. A task may contain one or more threads.
- A **thread** is the basic unit of execution, and must run in the context of a task (which provides the address space). All threads within a task share the tasks' resources (ports, memory, and so on). There is no notion of a "process" in Mach. Rather, a traditional process would be implemented as a task with a single thread of control.
- A **port** is the basic object reference mechanism in Mach, and is implemented as a kernel-protected communication channel. Communication is accomplished by sending messages to ports; messages are queued at the destination port if no thread is immediately ready to receive them. Ports are protected by kernel-managed capabilities, or *port rights*; a task must have a port right to send a message to a port. The programmer invokes an operation on an object by *sending* a message to a port associated with that object. The object being represented by a port *receives* the messages.

- A **port set** is a group of ports sharing a common message queue. A thread can receive messages for a port set, and thus service multiple ports. Each received message identifies the individual port (within the set) that it was received from; the receiver can use this to identify the object referred to by the message.
- A **message** is the basic method of communication between threads in Mach. It is a typed collection of data objects; for each object, it may contain the actual data or a pointer to out-of-line data. Port rights are passed in messages; passing port rights in messages is the only way to move them among tasks. (Passing a port right in shared memory does not work, because the Mach kernel will not permit the new task to use a right obtained in this manner.)
- A **memory object** is a source of memory; tasks may access it by mapping portions (or the entire object) into their address spaces. The object may be managed by a user-mode *external memory manager*. One example is a file managed by a file server; however, a memory object can be any object for which memory-mapped access makes sense. A mapped buffer implementation of a UNIX pipe is one example.

Figure 20.2 illustrates these abstractions, which we shall elaborate in the remainder of this chapter.

An unusual feature of Mach, and a key to the system's efficiency, is the blending of memory and interprocess-communication features. Whereas some other distributed systems (such as Solaris, with its NFS features) have special-purpose extensions to the file system to extend it over a network, Mach provides a general-purpose, extensible merger of memory and messages at the heart of its kernel. This feature not only allows Mach to be used for distributed and parallel programming, but also helps in the implementation of the kernel itself.

Mach connects memory management and communication (IPC) by allowing each to be used in the implementation of the other. Memory management is based on the use of *memory objects*. A memory object is represented by a port (or ports), and IPC messages are sent to this port to request operations (for example, *pagein, pageout*) on the object. Because IPC is used, memory objects may reside on remote systems and be accessed transparently. The kernel caches the contents of memory objects in local memory. Conversely, memory-management techniques are used in the implementation of message passing. Where possible, Mach passes messages by moving pointers to shared memory objects, rather than by copying the object itself.

IPC tends to involve considerable system overhead and is generally less efficient than is communication accomplished through shared memory, for intrasystem messages. Because Mach is a message-based kernel, it is important that message handling be carried out efficiently. Most of the

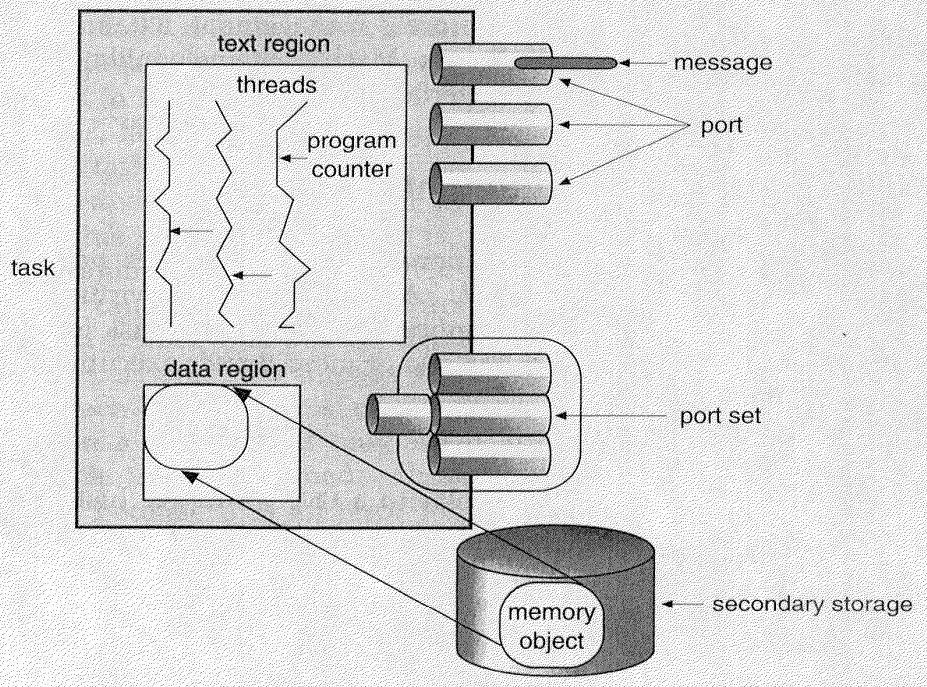

Figure 20.2 Mach's basic abstractions.

inefficiency of message handling in traditional operating systems is due to either the copying of messages from one task to another (if the message is intracomputer) or low network transfer speed (for intercomputer messages). To solve these problems, Mach uses virtual-memory remapping to transfer the contents of large messages. In other words, the message transfer modifies the receiving task's address space to include a copy of the message contents. *Virtual copy*, or *copy-on-write*, techniques are used to avoid or delay the actual copying of the data. There are several advantages to this approach:

- Increased flexibility in memory management to user programs
- Greater generality, allowing the virtual copy approach to be used in tightly and loosely coupled computers
- Improved performance over UNIX message passing
- Easier task migration; because ports are location independent, a task and all its ports can be moved from one machine to another; all tasks that previously communicated with the moved task can continue to do so because they reference a task by only its ports and communicate via messages to these ports

We shall detail the operation of process management, IPC, and memory management. Then, we shall discuss Mach's chameleonlike ability to support multiple operating-system interfaces.

20.4 ■ Process Management

A *task* can be thought of as a traditional process that does not have an instruction pointer or a register set. A task contains a virtual address space, a set of port rights, and accounting information. A task is a passive entity that does nothing unless it has one or more *threads* executing in it.

20.4.1 Basic Structure

A task containing one thread is similar to a UNIX process. Just as a **fork** system call produces a new UNIX process, Mach creates a new task to emulate this behavior. The new task's memory is a duplicate of the parent's address space, as dictated by the *inheritance attributes* of the parent's memory. The new task contains one thread, which is started at the same point as the creating **fork** call in the parent. Threads and tasks may also be suspended and resumed.

Threads are especially useful in server applications, which are common in UNIX, since one task can have multiple threads to service multiple requests to the task. They also allow efficient use of parallel computing resources. Rather than having one process on each processor (with the corresponding performance penalty and operating-system overhead), a task may have its threads spread among parallel processors. Threads also add efficiency to user-level programs. For instance, in UNIX, an entire process must wait when a page fault occurs, or when a system call is executed. In a task with multiple threads, only the thread that causes the page fault or executes a system call is delayed; all other threads continue executing. Because Mach has kernel-supported threads (see Section 4.5), the threads have some cost associated with them. They must have supporting data structures in the kernel, and more complex kernel-scheduling algorithms must be provided. These algorithms and thread states are discussed in Section 4.5.

At the user level, threads may be in one of two states.

- **Running:** The thread is either executing or waiting to be allocated a processor. A thread is considered to be running even if it is blocked within the kernel (waiting for a page fault to be satisfied, for instance).
- **Suspended:** The thread is neither executing on a processor nor waiting to be allocated a processor. A thread can resume its execution only if it is returned to the running state.

These two states are also associated with a task. An operation on a task affects all threads in a task, so suspending a task involves suspending all the threads in it. Task and thread suspensions are separate, independent mechanisms, however, so resuming a thread in a suspended task does not resume the task.

Mach provides primitives from which thread-synchronization tools can be built. This primitives provision is consistent with Mach's philosophy of providing minimum yet sufficient functionality in the kernel. The Mach IPC facility can be used for synchronization, with processes exchanging messages at rendezvous points. Thread-level synchronization is provided by calls to start and stop threads at appropriate times. A *suspend count* is kept for each thread. This count allows multiple suspend calls to be executed on a thread, and only when an equal number of resume calls occur is the thread resumed. Unfortunately, this feature has its own limitation. Because it is an error for a start call to be executed before a stop call (the suspend count would become negative), these routines cannot be used to synchronize shared data access. However, *wait* and *signal* operations associated with semaphores, and used for synchronization, can be implemented via the IPC calls. This method will be discussed in Section 20.5.

20.4.2 The C Threads Package

Mach provides low-level but flexible routines instead of polished, large, and more restrictive functions. Rather than making programmers work at this low level, Mach provides many higher-level interfaces for programming in C and other languages. For instance, the C *Threads* package provides multiple threads of control, shared variables, mutual exclusion for critical sections, and condition variables for synchronization. In fact, C Threads is one of the major influences of the POSIX P Threads standard, which many operating systems are being modified to support. As a result there are strong similarities between the C Threads and P Threads programming interfaces. The thread-control routines include calls to perform these tasks:

- Create a new thread within a task, given a function to execute and parameters as input. The thread then executes concurrently with the creating thread, which receives a thread identifier when the call returns.
- Destroy the calling thread, and return a value to the creating thread.
- Wait for a specific thread to terminate before allowing the calling thread to continue. This call is a synchronization tool, much like the UNIX **wait** system calls.

- Yield use of a processor, signaling that the scheduler may run another thread at this point. This call is also useful in the presence of a preemptive scheduler, as it can be used to relinquish the CPU voluntarily before the time quantum (scheduling interval) expires if a thread has no use for the CPU.

Mutual exclusion is achieved through the use of spinlocks, as were discussed in Chapter 6. The routines associated with mutual exclusion are these:

- The routine *mutex_alloc* dynamically creates a mutex variable.
- The routine *mutex_free* deallocates a dynamically created mutex variable.
- The routine *mutex_lock* locks a mutex variable. The executing thread loops in a spinlock until the lock is attained. A deadlock results if a thread with a lock tries to lock the same mutex variable. Bounded waiting is not guaranteed by the C Threads package. Rather, it is dependent on the hardware instructions used to implement the mutex routines.
- The routine *mutex_unlock* unlocks a mutex variable, much like the typical *signal* operation of a semaphore.

General synchronization without busy waiting can be achieved through the use of *condition variables,* which can be used to implement a *condition critical region* or a *monitor,* as was described in Chapter 6. A condition variable is associated with a mutex variable, and reflects a Boolean state of that variable. The routines associated with general synchronization are these:

- The routine *condition_alloc* dynamically allocates a condition variable.
- The routine *condition_free* deletes a dynamically created condition variable allocated as result of *condition_alloc*.
- The routine *condition_wait* unlocks the associated mutex variable, and blocks the thread until a *condition_signal* is executed on the condition variable, indicating that the event being waited for may have occurred. The mutex variable is then locked, and the thread continues. A *condition_signal* does not guarantee that the condition still holds when the unblocked thread finally returns from its *condition_wait* call, so the awakened thread must loop, executing the *condition_wait* routine until it is unblocked and the condition holds.

As an example of the C Threads routines, consider the bounded-buffer synchronization problem of Section 6.5.1. The producer and consumer are

```
repeat
        ...
    produce an item into nextp
        ...
    mutex_lock(mutex);
    while(full)
        condition_wait(nonfull, mutex);
        ...
    add nextp to buffer
        ...
    condition_signal(nonempty);
    mutex_unlock(mutex);
until false;
```

Figure 20.3 The structure of the producer process.

represented as threads that access the common bounded-buffer pool. We use a mutex variable to protect the buffer while it is being updated. Once we have exclusive access to the buffer, we use condition variables to block the producer thread if the buffer is full, and to block the consumer thread if the buffer is empty. Although this program normally would be written in the C language on a Mach system, we shall use the familiar Pascal-like syntax of previous chapters for clarity. As in Chapter 6, we assume that the buffer consists of n slots, each capable of holding one item. The *mutex* semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The *empty* and *full* semaphores count the number of empty and full buffers, respectively. The semaphore *empty* is initialized to the value n; the semaphore *full* is initialized to the value 0. The condition variable *nonempty* is true while the buffer has items in it, and *nonfull* is true if the buffer has an empty slot.

The first step includes the allocation of the mutex and condition variables:

```
mutex_alloc(mutex);
condition_alloc(nonempty, nonfull);
```

The code for the producer thread is shown in Figure 20.3; the code for the consumer thread is shown in Figure 20.4. When the program terminates, the mutex and condition variables need to be deallocated:

```
mutex_free(mutex);
condition_free(nonempty, nonfull);
```

```
repeat
    mutex_lock(mutex);
    while(empty)
        condition_wait(nonempty, mutex);
    ...
    remove an item from the buffer to nextc
    ...
    condition_signal(nonfull);
    mutex_unlock(mutex);
    ...
    consume the item in nextc
    ...
until false;
```

Figure 20.4 The structure of the consumer process.

20.4.3 The CPU Scheduler

The CPU scheduler for a thread-based multiprocessor operating system is more complex than are its process-based relatives. There are generally more threads in a multithreaded system than there are processes in a multitasking system. Keeping track of multiple processors is also difficult, and is a relatively new area of research. Mach uses a simple policy to keep the scheduler manageable. Only threads are scheduled, so no knowledge of tasks is needed in the scheduler. All threads compete equally for resources, including time quanta.

Each thread has an associated priority number ranging from 0 through 127, which is based on the exponential average of its usage of the CPU. That is, a thread that recently used the CPU for a large amount of time has the lowest priority. Mach uses the priority to place the thread in one of 32 global run queues. These queues are searched in priority order for waiting threads when a processor becomes idle. Mach also keeps per-processor, or local, run queues. A local run queue is used for threads that are bound to an individual processor. For instance, a device driver for a device connected to an individual CPU must run on only that CPU.

Instead of there being a central dispatcher that assigns threads to processors, each processor consults the local and global run queues to select the appropriate next thread to run. Threads in the local run queue have absolute priority over those in the global queues, because it is assumed that they are performing some chore for the kernel. The run queues (like most other objects in Mach) are locked when they are modified to avoid simultaneous changes by multiple processors. To speed dispatching of threads on the global run queue, Mach maintains a list of idle processors.

Additional scheduling difficulties arise from the multiprocessor nature of Mach. A fixed time quantum is not appropriate because there may be fewer runable threads than there are available processors, for instance. It would be wasteful to interrupt a thread with a context switch to the kernel when that thread's quantum runs out, only to have the thread be placed right back in the running state. Thus, instead of using a fixed-length quantum, Mach varies the size of the time quantum inversely with the total number of threads in the system. It keeps the time quantum over the entire system constant, however. For example, in a system with 10 processors, 11 threads, and a 100-millisecond quantum, a context switch needs to occur on each processor only once per second to maintain the desired quantum.

Of course, there are still complications to be considered. Even relinquishing the CPU while waiting for a resource is more difficult than it is on traditional operating systems. First, a call must be issued by a thread to alert the scheduler that the thread is about to block. This alert avoids race conditions and deadlocks, which could occur when the execution takes place in a multiprocessor environment. A second call actually causes the thread to be moved off the run queue until the appropriate event occurs. There are many other internal thread states that are used by the scheduler to control thread execution.

20.4.4 Exception Handling

Mach was designed to provide a single, simple, consistent exception-handling system, with support for standard as well as user-defined exceptions. To avoid redundancy in the kernel, Mach uses kernel primitives whenever possible. For instance, an exception handler is just another thread in the task in which the exception occurs. Remote procedure call (RPC) messages are used to synchronize the execution of the thread causing the exception (the "victim") and that of the handler, and to communicate information about the exception between the victim and handler. Mach exceptions are also used to emulate the BSD *signal* package, as described later in this section.

Disruptions to normal program execution come in two varieties: internally generated *exceptions* and external *interrupts*. Interrupts are asynchronously generated disruptions of a thread or task, whereas exceptions are caused by the occurrence of unusual conditions during a thread's execution. Mach's general-purpose exception facility is used for error detection and debugger support. This facility is also useful for other reasons, such as taking a core dump of a bad task, allowing tasks to handle their own errors (mostly arithmetic), and emulating instructions not implemented in hardware.

Mach supports two different granularities of exception handling. Error handling is supported by per-thread exception handling, whereas

debuggers use per-task handling. It makes little sense to try to debug only one thread, or to have exceptions from multiple threads invoke multiple debuggers. Aside from this distinction, the only other difference between the two types of exceptions lies in their inheritance from a parent task. Taskwide exception-handling facilities are passed from the parent to child tasks, so debuggers are able to manipulate an entire tree of tasks. Error handlers are not inherited, and default to no handler at thread- and task-creation time. Finally, error handlers take precedence over debuggers if the exceptions occur simultaneously. The reason for this approach is that error handlers are normally part of the task, and therefore should execute normally even in the presence of a debugger.

Exception handling proceeds as follows:

- The victim thread causes notification of an exception's occurrence via a *raise* RPC message being sent to the handler.
- The victim then calls a routine to wait until the exception is handled.
- The handler receives notification of the exception, usually including information about the exception, the thread, and the task causing the exception.
- The handler performs its function according to the type of exception. The handler's action involves *clearing* the exception, causing the victim to resume, or *terminating* the victim thread.

To support the execution of BSD programs, Mach needs to support BSD-style signals. Signals provide software generated interrupts and exceptions. Unfortunately, signals are of limited functionality in multithreaded operating systems. The first problem is that, in UNIX, a signal's handler must be a routine in the process receiving the signal. If the signal is caused by a problem in the process itself (for example, a division by zero), the problem cannot be remedied, because a process has limited access to its own context. A second, more troublesome aspect of signals is that they were designed for only single-threaded programs. For instance, it makes no sense for all threads in a task to get a signal, but how can a signal be seen by only one thread?

Because the signal system must work correctly with multithreaded applications for Mach to run 4.3BSD programs, signals could not be abandoned. Producing a functionally correct signal package required several rewrites of the code, however! A final problem with UNIX signals is that they can be lost. This loss occurs when another signal of the same type occurs before the first is handled. Mach exceptions are queued as a result of their RPC implementation.

Externally generated signals, including those sent from one BSD process to another, are processed by the BSD server section of the Mach 2.5 kernel.

Their behavior is therefore the same as it is under BSD. Hardware exceptions are a different matter, because BSD programs expect to receive hardware exceptions as signals. Therefore, a hardware exception caused by a thread must arrive at the thread as a signal. So that this result is produced, hardware exceptions are converted to exception RPCs. For tasks and threads that do not make explicit use of the Mach exception-handling facility, the destination of this RPC defaults to an in-kernel task. This task has only one purpose: Its thread runs in a continuous loop, receiving these exception RPCs. For each RPC, it converts the exception into the appropriate signal, which is sent to the thread that caused the hardware exception. It then completes the RPC, clearing the original exception condition. With the completion of the RPC, the initiating thread reenters the run state. It immediately sees the signal and executes its signal-handling code. In this manner, all hardware exceptions begin in a uniform way — as exceptions RPCs. Threads not designed to handle such exceptions, however, receive the exceptions as they would on a standard BSD system — as signals. In Mach 3.0, the signal-handling code is moved entirely into a server, but the overall structure and flow of control is similar to those of Mach 2.5.

20.5 ■ Interprocess Communication

Most commercial operating systems, such as UNIX, provide communication between processes, and between hosts with fixed, global names (internet addresses). There is no location independence of facilities, because any remote system needing to use a facility must know the name of the system providing that facility. Usually, data in the messages are untyped streams of bytes. Mach simplifies this picture by sending messages between location-independent ports. The messages contain typed data for ease of interpretation. All BSD communication methods can be implemented with this simplified system.

The two components of Mach IPC are *ports* and *messages*. Almost everything in Mach is an object, and all objects are addressed via their communications ports. Messages are sent to these ports to initiate operations on the objects by the routines that implement the objects. By depending on only ports and messages for all communication, Mach delivers location independence of objects and security of communication. Data independence is provided by the NetMsgServer task, as discussed later. Mach ensures security by requiring that message senders and receivers have *rights*. A right consists of a port name and a capability (send or receive) on that port, and is much like a capability in object-oriented systems. There can be only one task with receive rights to any given port, but many tasks may have send rights. When an object is created, its creator also allocates a port to represent the object, and obtains

the access rights to that port. Rights can be given out by the creator of the object (including the kernel), and are passed in messages. If the holder of a receive right sends that right in a message, the receiver of the message gains the right and the sender loses it. A task may allocate ports to allow access to any objects it owns, or for communication. The destruction of either a port or the holder of the receive right causes the revocation of all rights to that port, and the tasks holding send rights can be notified if desired.

20.5.1 Ports

A port is implemented as a protected, bounded queue within the kernel of the system on which the object resides. If a queue is full, a sender may abort the send, wait for a slot to become available in the queue, or have the kernel deliver the message for it.

There are several system calls to provide the port functionality:

- Allocate a new port in a specified task and give the caller's task all access rights to the new port. The port name is returned.
- Deallocate a task's access rights to a port. If the task holds the receive right, the port is destroyed and all other tasks with send rights are, potentially, notified.
- Get the current status of a task's port.
- Create a backup port, which is given the receive right for a port if the task containing the receive right requests its deallocation (or terminates).

When a task is created, the kernel creates several ports for it. The function *task_self* returns the name of the port that represents the task in calls to the kernel. For instance, for a task to allocate a new port, it would call *port_allocate* with *task_self* as the name of the task that will own the port. Thread creation results in a similar *thread_self* thread kernel port. This scheme is similar to the standard process-id concept found in UNIX. Another port created for a task is returned by *task_notify*, and is the name of the port to which the kernel will send event-notification messages (such as notifications of port terminations).

Ports can also be collected into *port sets*. This facility is useful if one thread is to service requests coming in on multiple ports (for example, for multiple objects). A port may be a member of at most one port set at a time, and, if a port is in a set, it may not be used directly to receive messages. Instead, the message will be routed to the port set's queue. A port set may not be passed in messages, unlike a port. Port sets are objects that serve a purpose similar to the 4.3BSD **select** system call, but they are more efficient.

20.5.2 Messages

A message consists of a fixed-length header and a variable number of typed data objects. The header contains the destination's port name, the name of the reply port to which return messages should be sent, and the length of the message (see Figure 20.5). The data in the message (*in-line* data) were limited to less than 8K in Mach 2.5 systems, but Mach 3.0 has no limit. Any data exceeding that limit must be sent in multiple messages, or more likely via reference by a pointer in a message (*out-of-line* data, as we shall describe shortly). Each data section may be a simple type (numbers or characters), port rights, or pointers to out-of-line data. Each section has an associated type, so that the receiver can unpack the data correctly even if it uses a byte ordering different from that used by the sender. The kernel also inspects the message for certain types of data. For instance, the kernel must process port information within a message, either by translating the port name into an internal port data structure address, or by forwarding it for processing to the NetMsgServer, as we shall explain.

The use of pointers in a message provides the means to transfer the entire address space of a task in one single message. The kernel also must process pointers to out-of-line data, as a pointer to data in the sender's address space would be invalid in the receiver's — especially if the sender and receiver reside on different systems! Generally, systems send messages

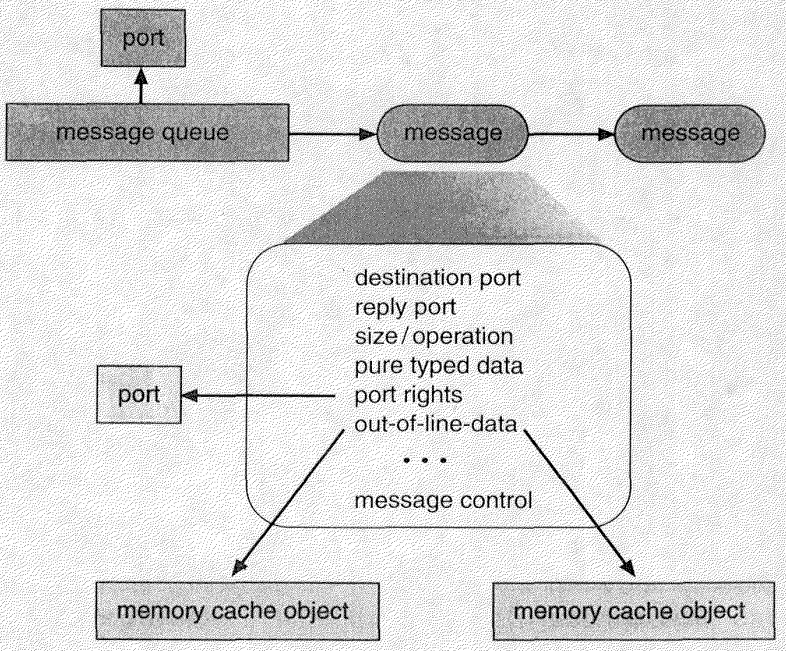

Figure 20.5 Mach messages.

by copying the data from the sender to the receiver. Because this technique can be inefficient, especially in the case of large messages, Mach optimizes this procedure. The data referenced by a pointer in a message being sent to a port on the same system are not copied between the sender and the receiver. Instead, the address map of the receiving task is modified to include a copy-on-write copy of the pages of the message. This operation is *much* faster than a data copy, and makes message passing efficient. In essence, message passing is implemented via virtual-memory management.

In Version 2.5, this operation was implemented in two phases. A pointer to a region of memory caused the kernel to map that region of memory into its own space temporarily, setting the sender's memory map to copy-on-write mode to ensure that any modifications did not affect the original version of the data. When a message was received at its destination, the kernel moved its mapping to the receiver's address space, using a newly allocated region of virtual memory within that task.

In Version 3, this process was simplified. The kernel creates a data structure that would be a copy of the region if it were part of an address map. On receipt, this data structure is added to the receiver's map and becomes a copy accessible to the receiver.

The newly allocated regions in a task do not need to be contiguous with previous allocations, so Mach virtual memory is said to be *sparse*, consisting of regions of data separated by unallocated addresses. A full message transfer is shown in Figure 20.6.

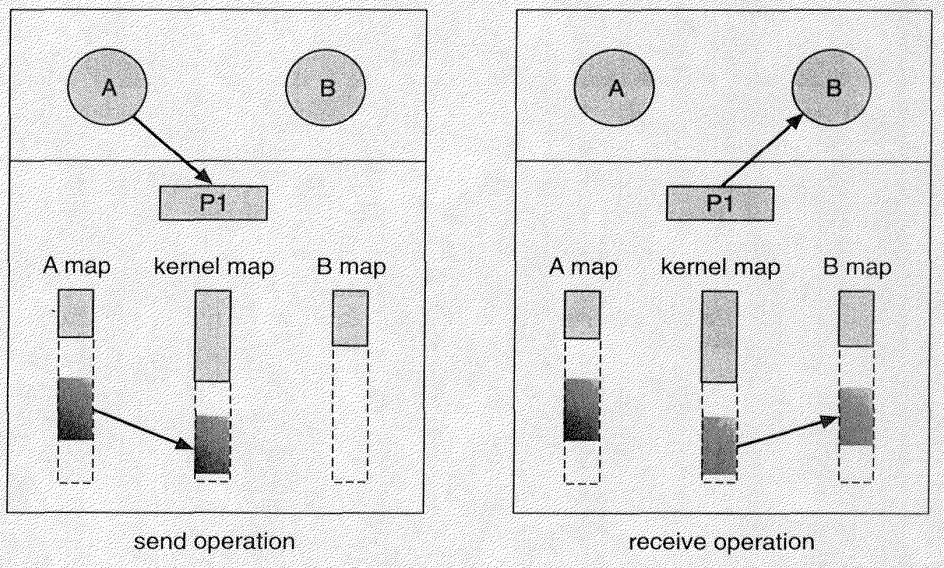

Figure 20.6 Mach message transfer.

20.5.3 The NetMsgServer

For a message to be sent between computers, the destination of a message must be located, and the message must be transmitted to the destination. UNIX traditionally leaves these mechanisms to the low-level network protocols, which require the use of statically assigned communication endpoints (for example, the port number for services based on TCP or UDP). One of Mach's tenets is that all objects within the system are location independent, and that the location is transparent to the user. This tenet requires Mach to provide location-transparent naming and transport to extend IPC across multiple computers. This naming and transport are performed by the *Network Message Server* or NetMsgServer, a user-level capability-based networking daemon that forwards messages between hosts. It also provides a primitive networkwide name service that allows tasks to register ports for lookup by tasks on any other computer in the network. Mach ports can be transferred only in messages, and messages must be sent to ports; the primitive name service solves the problem of transferring the first port that allows tasks on different computers to exchange messages. Subsequent IPC interactions are fully transparent; the NetMsgServer tracks all rights and out-of-line memory passed in intercomputer messages, and arranges for the appropriate transfers. The NetMsgServers maintain among themselves a distributed database of port rights that have been transferred between computers and of the ports to which these rights correspond.

The kernel uses the NetMsgServer when a message needs to be sent to a port that is not on the kernel's computer. Mach's kernel IPC is used to transfer the message to the local NetMsgServer. The NetMsgServer then uses whatever network protocols are appropriate to transfer the message to its peer on the other computer; the notion of a NetMsgServer is protocol-independent, and NetMsgServers have been built that use various protocols. Of course, the NetMsgServers involved in a transfer must agree on the protocol used. Finally, the NetMsgServer on the destination computer uses that kernel's IPC to send the message to the correct destination task. The ability to extend local IPC transparently across nodes is supported by the use of proxy ports. When a send right is transferred from one computer to another, the NetMsgServer on the destination computer creates a new port, or proxy, to represent the original port at the destination. Messages sent to this proxy are received by the NetMsgServer and are forwarded transparently to the original port; this procedure is one example of how the NetMsgServers cooperate to make a proxy indistinguishable from the original port.

Because Mach is designed to function in a network of heterogeneous systems, it must provide a way to send between systems data that are formatted in a way that is understandable by both the sender and receiver. Unfortunately, computers vary the format in which they store types of

data. For instance, an integer on one system might take 2 bytes to store, and the most significant byte might be stored before the least significant one. Another system might reverse this ordering. The NetMsgServer therefore uses the type information stored in a message to translate the data from the sender's to the receiver's format. In this way, all data are represented correctly when they reach their destination.

The NetMsgServer on a given computer accepts RPCs that add, look up, and remove network ports from the NetMsgServer's name service. As a security precaution, a port value provided in an add request must match that in the remove request for a thread to ask for a port name to be removed from the database.

As an example of the NetMsgServer's operation, consider a thread on node A sending a message to a port that happens to be in a task on node B. The program simply sends a message to a port to which it has a send right. The message is first passed to the kernel, which delivers it to its first recipient, the NetMsgServer on node A. The NetMsgServer then contacts (through its database information) the NetMsgServer on node B and sends the message. The NetMsgServer on node B then presents the message to the kernel with the appropriate local port for node B. The kernel finally provides the message to the receiving task when a thread in that task executes a *msg_receive* call. This sequence of events is shown in Figure 20.7.

Mach 3.0 provides an alternative to the NetMsgServer as part of its improved support for NORMA multiprocessors. The NORMA IPC subsystem

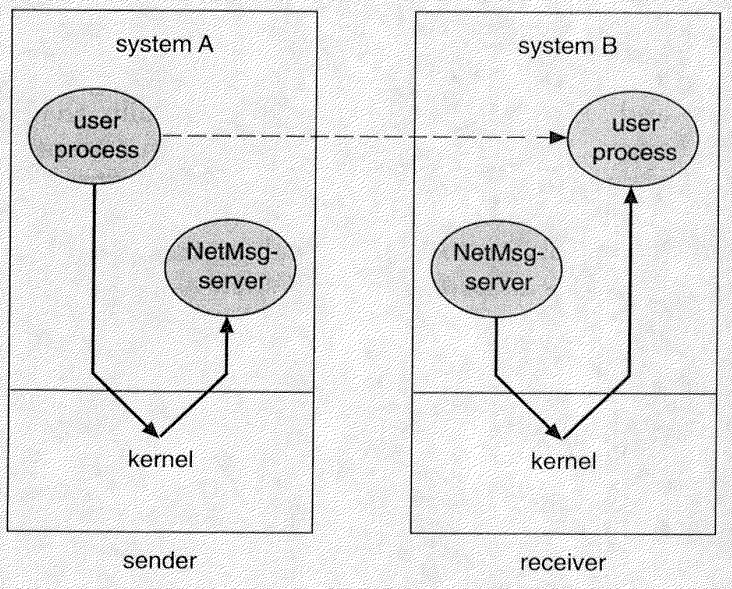

Figure 20.7 Network IPC forwarding by NetMsgServer.

of Mach 3.0 implements functionality similar to the NetMsgServer directly in the Mach kernel, providing much more efficient internode IPC for multicomputers with fast interconnection hardware. For example, the time-consuming copying of messages between the NetMsgServer and the kernel is eliminated. Use of NORMA IPC does not exclude use of the NetMsgServer; the NetMsgServer can still be used to provide MACH IPC service over networks that link a NORMA multiprocessor to other computers. In addition to NORMA IPC, Mach 3.0 also provides support for memory management across a NORMA system, and the ability for a task in such a system to create child tasks on nodes other than its own. These features support the implementation of a single-system-image operating system on a NORMA multiprocessor; the multiprocessor behaves like one large system, rather than like an assemblage of smaller systems (for both users and applications).

20.5.4 Synchronization Through IPC

The IPC mechanism is extremely flexible, and is used throughout Mach. For example, it may be used for thread synchronization. A port may be used as a synchronization variable, and may have n messages sent to it for n resources. Any thread wishing to use a resource executes a receive call on that port. The thread will receive a message if the resource is available; otherwise, it will wait on the port until a message is available there. To return a resource after use, the thread can send a message to the port. In this regard, receiving is equivalent to the semaphore operation *wait*, and sending is equivalent to *signal*. This method can be used for synchronizing semaphore operations among threads in the same task, but cannot be used for synchronization among tasks, because only one task may have receive rights to a port. For more general-purpose semaphores, a simple daemon may be written that implements the same method.

20.6 ■ Memory Management

Given the object-oriented nature of Mach, it is not surprising that a principle abstraction in Mach is the *memory object*. Memory objects are used to manage secondary storage, and generally represent files, pipes, or other data that are mapped into virtual memory for reading and writing (Figure 20.8). Memory objects may be backed by user-level memory managers, which take the place of the more traditional kernel-incorporated virtual-memory pager found in other operating systems. In contrast to the traditional approach of having the kernel provide management of secondary storage, Mach treats secondary-storage objects (usually files) as it does all other objects in the system. Each object has a port associated

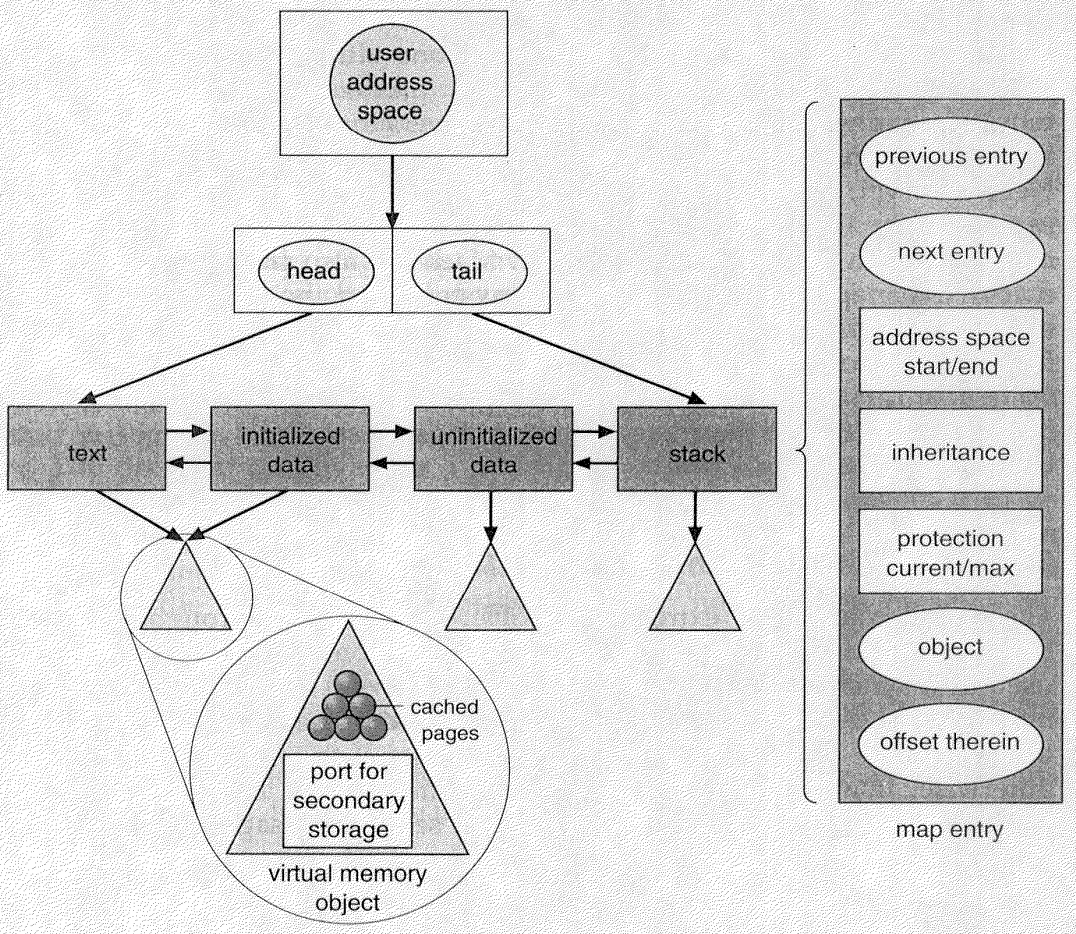

Figure 20.8 Mach virtual memory task address map.

with it, and may be manipulated by messages being sent to its port. Memory objects — unlike the memory-management routines in monolithic, traditional kernels — allow easy experimentation with new memory-manipulation algorithms.

20.6.1 Basic Structure

The virtual address space of a task is generally *sparse*, consisting of many holes of unallocated space. For instance, a memory-mapped file is placed in some set of addresses. Large messages are also transferred as shared memory segments. For each of these segments, a section of virtual-memory address is used to provide the threads with access to the message. As new items are mapped or removed from the address space, holes of unallocated memory appear in the address space.

Mach makes no attempt to compress the address space, although a task may fail (crash) if it has no room for a requested region in its address space. Given that address spaces are 4 gigabytes or more, this limitation is not currently a problem. However, maintaining a regular page table for a 4 gigabyte address space for each task, especially one with holes in it, would use excessive amounts of memory (1 megabyte or more). The key to sparse address spaces is that page-table space is used for only currently allocated regions. When a page fault occurs, the kernel must check to see whether the page is in a valid region, rather than simply indexing into the page table and checking the entry. Although the resulting lookup is more complex, the benefits of reduced memory-storage requirements and simpler address-space maintenance make the approach worthwhile.

Mach also has system calls to support standard virtual-memory functionality, including the allocation, deallocation, and copying of virtual memory. When allocating a new virtual-memory object, the thread may provide an address for the object or may let the kernel choose the address. Physical memory is not allocated until pages in this object are accessed. The object's backing store is managed by the default pager (discussed in Section 20.6.2). Virtual-memory objects are also allocated automatically when a task receives a message containing out-of-line data.

Associated system calls return information about a memory object in a task's address space, change the access protection of the object, and specify how an object is to be passed to child tasks at the time of their creation (shared, copy-on-write, or not present).

20.6.2 User-Level Memory Managers

A secondary-storage object is usually mapped into the virtual address space of a task. Mach maintains a cache of memory-resident pages of all mapped objects, as in other virtual-memory implementations. However, a page fault occurring when a thread accesses a nonresident page is executed as a message to the object's port. The concept of a memory object being created and serviced by nonkernel tasks (unlike threads, for instance, which are created and maintained by only the kernel) is important. The end result is that, in the traditional sense, memory can be paged by user-written memory managers. When the object is destroyed, it is up to the memory manager to write back any changed pages to secondary storage. No assumptions are made by Mach about the content or importance of memory objects, so the memory objects are independent of the kernel.

There are several circumstances in which user-level memory managers are insufficient. For instance, a task allocating a new region of virtual memory might not have a memory manager assigned to that region, since it does not represent a secondary-storage object (but must be paged), or a memory manager could fail to perform pageout. Mach itself also needs a

memory manager to take care of its memory needs. For these cases, Mach provides a *default memory manager*. The Mach 2.5 default memory manager uses the standard file system to store data that must be written to disk, rather than requiring a separate swap space, as in 4.3BSD. In Mach 3.0 (and OSF/1), the default memory manager is capable of using either files in a standard filesystem or dedicated disk partitions. The default memory manager has an interface similar to that of the user-level ones, but with some extensions to support its role as the memory manager that can be relied on to perform pageout when user-level managers fail to do so.

Pageout policy is implemented by an internal kernel thread, the *pageout daemon*. A paging algorithm based on FIFO with second chance (Section 9.5.4) is used to select pages for replacement. The selected pages are sent to the appropriate manager (either user level or default) for actual pageout. A user-level manager may be more intelligent than the default manager, and may implement a different paging algorithm suitable to the object it is backing (that is, by selecting some other page and forcibly paging it out). If a user-level manager fails to reduce the resident set of pages when asked to do so by the kernel, the default memory manager is invoked and it pages out the user-level manager to reduce the user-level manager's resident set size. Should the user-level manager recover from the problem that prevented it from performing its own pageouts, it will touch these pages (causing the kernel to page them in again), and can then page them out as it sees fit.

If a thread needs access to data in a memory object (for instance, a file), it invokes the *vm_map* system call. Included in this system call is a port which identifies the object, and the memory manager which is responsible for the region. The kernel executes calls on this port when data are to be read or written in that region. An added complexity is that the kernel makes these calls asynchronously, since it would not be reasonable for the kernel to be waiting on a user-level thread. Unlike the situation with pageout, the kernel has no recourse if its request is not satisfied by the external memory manager. The kernel has no knowledge of the contents of an object or of how that object must be manipulated.

Memory managers are responsible for the consistency of the contents of a memory object mapped by tasks on different machines (tasks on a single machine share a single copy of a mapped memory object). Consider a situation in which tasks on two different machines attempt to modify the same page of an object concurrently. It is up to the manager to decide whether these modifications must be serialized. A conservative manager implementing strict memory consistency would force the modifications to be serialized by granting write access to only one kernel at a time. A more sophisticated manager could allow both accesses to proceed concurrently (for example, if the manager knew that the two tasks were modifying distinct areas within the page, and that it could merge the modifications successfully at some future time). Note that most external memory

managers written for Mach (for example, those implementing mapped files) do not implement logic for dealing with multiple kernels, due to the complexity of such logic.

When the first *vm_map* call is made on a memory object, the kernel sends a message to the memory manager port passed in the call, invoking the *memory_manager_init* routine, which the memory manager must provide as part of its support of a memory object. The two ports passed to the memory manager are a *control port* and a *name port*. The control port is used by the memory manager to provide data to the kernel (for example, pages to be made resident). Name ports are used throughout Mach. They do not receive messages, but rather are used simply as a point of reference and comparison. Finally, the memory object must respond to a *memory_manager_init* call with a *memory_object_set_attributes* call to indicate that it is ready to accept requests. When all tasks with send rights to a memory object relinquish those rights, the kernel deallocates the object's ports, thus freeing the memory manager and memory object for destruction.

There are several kernel calls that are needed to support external memory managers. The *vm_map* call has already been discussed in the paragraph above. There are also commands to get and set attributes and to provide page-level locking when it is required (for instance, after a page fault has occurred but before the memory manager has returned the appropriate data). Another call is used by the memory manager to pass a page (or multiple pages, if read-ahead is being used) to the kernel in response to a page fault. This call is necessary since the kernel invokes the memory manager asynchronously. There are also several calls to allow the memory manager to report errors to the kernel.

The memory manager itself must provide support for several calls so that it can support an object. We have already discussed *memory_object_init* and others. When a thread causes a page fault on a memory object's page, the kernel sends a *memory_object_data_request* to the memory object's port on behalf of the faulting thread. The thread is placed in wait state until the memory manager either returns the page in a *memory_object_data_provided* call, or returns an appropriate error to the kernel. Any of the pages that have been modified, or any *precious pages* that the kernel needs to remove from resident memory (due to page aging, for instance), are sent to the memory object via *memory_object_data_write*. Precious pages are pages that may not have been modified, but that cannot be discarded as they otherwise would, because the memory manager no longer retains a copy. The memory manager declares these pages to be precious and expects the kernel to return them when they are removed from memory. Precious pages save unnecessary duplication and copying of memory.

Again, there are several other calls for locking, protection information and modification, and the other details with which all virtual memory systems must deal.

In the current version, Mach does not allow external memory managers to affect the page-replacement algorithm directly. Mach does not export the memory-access information that would be needed for an external task to select the least recently used page, for instance. Methods of providing such information are currently under investigation. An external memory manager is still useful for a variety of reasons, however:

- It may reject the kernel's replacement victim if it knows of a better candidate (for instance, MRU page replacement).
- It may monitor the memory object it is backing, and request pages to be paged out before the memory usage invokes Mach's pageout daemon.
- It is especially important in maintaining consistency of secondary storage for threads on multiple processors, as we shall show in Section 20.6.3.
- It can control the order of operations on secondary storage, to enforce consistency constraints demanded by database management systems. For example, in transaction logging, transactions must be written to a log file on disk before they modify the database data.
- It can control mapped file access.

20.6.3 Shared Memory

Mach uses shared memory to reduce the complexity of various system facilities, as well as to provide these features in an efficient manner. Shared memory generally provides extremely fast interprocess communication, reduces overhead in file management, and helps to support multiprocessing and database management. Mach does not use shared memory for all these traditional shared-memory roles, however. For instance, all threads in a task share that task's memory, so no formal shared-memory facility is needed within a task. However, Mach must still provide traditional shared memory to support other operating-system constructs, such as the UNIX **fork** system call.

It is obviously difficult for tasks on multiple machines to share memory, and to maintain data consistency. Mach does not try to solve this problem directly; rather, it provides facilities to allow the problem to be solved. Mach supports consistent shared memory only when the memory is shared by tasks running on processors that share memory. A parent task is able to declare which regions of memory are to be *inherited* by its children, and which are to be readable–writable. This scheme is different from copy-on-write inheritance, in which each task maintains its own copy of any changed pages. A writable object is addressed from each task's address map, and all changes are made to the same copy. The threads

within the tasks are responsible for coordinating changes to memory so that they do not interfere with one another (by writing to the same location concurrently). This coordination may be done through normal synchronization methods: critical sections or mutual-exclusion locks.

For the case of memory shared among separate machines, Mach allows the use of external memory managers. If a set of unrelated tasks wishes to share a section of memory, the tasks may use the same external memory manager and access the same secondary-storage areas through it. The implementor of this system would need to write the tasks and the external pager. This pager could be as simple or as complicated as needed. A simple implementation would allow no readers while a page was being written to. Any write attempt would cause the pager to invalidate the page in all tasks currently accessing it. The pager would then allow the write and would revalidate the readers with the new version of the page. The readers would simply wait on a page fault until the page again became available. Mach provides such a memory manager: the Network Memory Server (NetMemServer). For multicomputers, the NORMA configuration of Mach 3.0 provides similar support as a standard part of the kernel. This XMM subsystem allows multicomputer systems to use external memory managers that do not incorporate logic for dealing with multiple kernels; the XMM subsystem is responsible for maintaining data consistency among multiple kernels that share memory, and makes these kernels appear to be a single kernel to the memory manager. The XMM subsystem also implements virtual copy logic for the mapped objects that it manages. This virtual copy logic includes both copy-on-reference among multicomputer kernels, and sophisticated copy-on-write optimizations.

20.7 ■ Programmer Interface

There are several levels at which a programmer may work within Mach. There is, of course, the system-call level, which, in Mach 2.5, is equivalent to the 4.3BSD system-call interface. Version 2.5 includes most of 4.3BSD as one thread in the kernel. A BSD system call traps to the kernel and is serviced by this thread on behalf of caller, much as standard BSD would handle it. The emulation is not multithreaded, so it has limited efficiency.

Mach 3.0 has moved from the single-server model to support of multiple servers. It has therefore become a true microkernel without the full features normally found in a kernel. Rather, full functionality can be provided via emulation libraries, servers, or a combination of the two. In keeping with the definition of a microkernel, the emulation libraries and servers run outside the kernel at user level. In this way, multiple operating systems can run concurrently on one Mach 3.0 kernel.

An emulation library is a set of routines that lives in a read-only part of a program's address space. Any operating-system calls the program makes

are translated into subroutine calls to the library. Single-user operating systems, such as MS-DOS and the Macintosh operating system, have been implemented solely as emulation libraries. For efficiency reasons, the emulation library lives in the address space of the program needing its functionality, but in theory could be a separate task.

More complex operating systems are emulated through the use of libraries and one or more servers. System calls that cannot be implemented in the library are redirected to the appropriate server. Servers can be multithreaded for improved efficiency. BSD and OSF/1 are implemented as single multithreaded servers. Systems can be decomposed into multiple servers for greater modularity.

Functionally, a system call starts in a task, and passes through the kernel before being redirected, if appropriate, to the library in the task's address space or to a server. Although this extra transfer of control will decrease the efficiency of Mach, this decrease is somewhat ameliorated by the ability for multiple threads to be executing BSD-like code concurrently.

At the next higher programming level is the *C Threads* package. This package is a run-time library that provides a C language interface to the basic Mach threads primitives. It provides convenient access to these primitives, including routines for the forking and joining of threads, mutual exclusion through mutex variables (Section 20.4.2), and synchronization through use of condition variables. Unfortunately, it is not appropriate for the C Threads package to be used between systems that share no memory (NORMA systems), since it depends on shared memory to implement its constructs. There is currently no equivalent of C Threads for NORMA systems. Other run-time libraries have been written for Mach, including threads support for other languages.

Although the use of primitives makes Mach flexible, it also makes many programming tasks repetitive. For instance, significant amounts of code are associated with sending and receiving messages in each task that uses messages (which, in Mach, is most tasks). The designers of Mach therefore provide an interface generator (or stub generator) called *MIG*. MIG is essentially a compiler that takes as input a definition of the interface to be used (declarations of variables, types and procedures), and generates the RPC interface code needed to send and receive the messages fitting this definition and to connect the messages to the sending and receiving threads.

20.8 ■ Summary

The Mach operating system is designed to incorporate the many recent innovations in operating-system research to produce a fully functional, technically advanced operating system.

The Mach operating system was designed with the following three critical goals in mind:

- Emulate 4.3BSD UNIX so that the executable files from a UNIX system can run correctly under Mach.
- Have a modern operating system that supports many memory models, and parallel and distributed computing.
- Design a kernel that is simpler and easier to modify than is 4.3BSD.

As we have shown in this chapter, Mach is well on its way to achieving these goals.

Mach 2.5 includes 4.3BSD in its kernel, which provides the emulation needed but enlarges the kernel. This 4.3BSD code has been rewritten to provide the same 4.3 functionality, but to use the Mach primitives. This change allows the 4.3BSD support code to run in user space on a Mach 3.0 system.

Mach uses lightweight processes, in the form of multiple threads of execution within one task (or address space), to support multiprocessing and parallel computation. Its extensive use of messages as the only communications method ensures that protection mechanisms are complete and efficient. By integrating messages with the virtual-memory system, Mach also ensures that messages can be handled efficiently. Finally, by having the virtual-memory system use messages to communicate with the daemons managing the backing store, Mach provides great flexibility in the design and implementation of these memory-object-managing tasks.

By providing low-level, or primitive, system calls from which more complex functions may be built, Mach reduces the size of the kernel while permitting operating-system emulation at the user level, much like IBM's virtual-machine systems.

■ Exercises

20.1 What three features of Mach make it appropriate for distributed processing?

20.2 Name two ways that port sets are useful in implementing parallel programs.

20.3 Consider an application that maintains a database of information, and provides facilities for other tasks to add, delete, and query the database. Give three configurations of ports, threads, and message types that could be used to implement this system. Which is the best? Explain your answer.

20.4 Give the outline of a task that would migrate subtasks (tasks it creates) to other systems. Include information about how it would decide when to migrate tasks, which tasks to migrate, and how the migration would take place.

20.5 Name two types of applications for which you would use the MIG package.

20.6 Why would someone use the low-level system calls, instead of the C Threads package?

20.7 Why are external memory managers not able to replace the internal page-replacement algorithms? What information would need to be made available to the external managers for them to make page-replacement decisions? Why would providing this information violate the principle behind the external managers?

20.8 Why is it difficult to implement mutual exclusion and condition variables in an environment where like-CPUs do not share any memory? What approach and mechanism could be used to make such features available on a NORMA system?

20.9 What are the advantages to rewriting the 4.3BSD code as an external, user-level library, rather than leaving it as part of the Mach kernel? Are there any disadvantages? Explain your answer.

Bibliographic Notes

The Accent operating system was described by Rashid and Robertson [1981]. An historical overview of the progression from an even earlier system, RIG, through Accent to Mach was given by Rashid [1986]. General discussions concerning the Mach model are offered by Tevanian and Smith [1989].

Accetta et al. [1986] presented an overview of the original design of Mach. The Mach scheduler was described in detail by Tevanian et al. [1987a] and Black [1990]. An early version of the Mach shared memory and memory-mapping system was presented by Tevanian et al. [1987b].

The most current description of the C Threads package appears in Cooper and Draves [1987]; MIG was described by Draves et al. [1989]. An overview of these packages' functionality and a general introduction to programming in Mach was presented by Walmer and Thompson [1989] and Boykin et al. [1993].

Black et al. [1988] discussed the Mach exception-handling facility. A multithreaded debugger based on this mechanism was described in Caswell and Black [1989].

A series of talks about Mach sponsored by the OSF UNIX consortium is available on videotape from OSF. Topics include an overview, threads, networking, memory management, many internal details, and some example implementations of Mach. The slides from these talks were given in [OSF 1989].

On systems where USENET News is available (most educational institutions in the United States, and some overseas), the news group *comp.os.mach* is used to exchange information on the Mach project and its components.

An overview of the microkernel structure of Mach 3.0, complete with performance analysis of Mach 2.5 and 3.0 compared to other systems, was given in Black et al. [1992]. Details of the kernel internals and interfaces of Mach 3.0 were provided in Loepere [1992]. Tanenbaum [1992] presented a comparison of Mach and Amoeba. Discussions concerning parallelization in Mach and 4.3BSD are offered by Boykin and Langerman [1990].

Ongoing research was presented at USENIX Mach and Micro-kernel Symposia [USENIX 1990, USENIX 1991, and USENIX 1992b]. Active research areas include virtual memory, real time, and security [McNamee and Armstrong 1990].

CHAPTER 21

HISTORICAL PERSPECTIVE

In Chapter 1, we presented a short historical survey of the development of operating systems. That survey was brief and lacked detail, since the fundamental concepts of operating systems (CPU scheduling, memory management, processes, and so on) had not yet been presented. By now, however, you understand the basic concepts. We are thus in a position to examine how our concepts have been applied in several older and highly influential operating systems. Some of them (such as the XDS-940 or the THE system) were one-of-a-kind systems; others (such as OS/360) are widely used. The order of presentation has been chosen to highlight the similarities and differences of the systems, and is not strictly chronological or ordered by importance. The serious student of operating systems should be familiar with all these systems.

The treatment of each system is brief, but each section contains references to further reading. The papers, written by the designers of the systems, are important both for their technical content, and for their style and flavor.

21.1 ■ Atlas

The Atlas operating system [Kilburn et al. 1961, Howarth et al. 1961] was designed at the University of Manchester in England in the late 1950s and early 1960s. Many of its basic features, which were novel at the time, have become standard parts of modern operating systems. Device drivers were a major part of the system. In addition, system calls were added by a set of special instructions called *extra codes*.

Atlas was a batch operating system with spooling. Spooling allowed the system to schedule jobs according to the availability of peripheral devices, such as magnetic tape units, paper tape readers, paper tape punches, line printers, card readers, or card punches.

The most remarkable feature of Atlas, however, was its memory management. Core memory was new and expensive at the time. Many computers, like the IBM 650, used a drum for primary memory. The Atlas system used a drum for its main memory, but had a small amount of core memory that was used as a cache for the drum. Demand paging was used to transfer information between core memory and the drum automatically.

The Atlas system used a British computer with 48-bit words. Addresses were 24 bits, but were encoded in decimal, which allowed only 1 million words to be addressed. At that time, this was a very large address space. The physical memory for Atlas was a 98K word drum and 16K words of core. Memory was divided into 512-word pages, providing 32 frames in physical memory. An associative memory of 32 registers implemented the mapping from a virtual address to a physical address.

If a page fault occurred, a page-replacement algorithm was invoked. One memory frame was always kept empty, so that a drum transfer could start immediately. The page-replacement algorithm attempted to predict the future memory-accessing behavior based on past behavior. A reference bit for each frame was set whenever the frame was accessed. The reference bits were read into memory every 1024 instructions. The last 32 values of the reference bit were used to define the time since the last reference (t_1) and the interval between the last two references (t_2). Pages were chosen for replacement in the following order:

1. Any page with $t_1 > t_2 + 1$. This page is considered to be no longer in use.
2. If $t_1 \leq t_2$ for all pages, then replace that page with the largest $t_2 - t_1$.

The page-replacement algorithm assumes that programs access memory in loops. If the time between the last two references is t_2, then another reference is expected t_2 time units later. If a reference does not occur ($t_1 > t_2$), it is assumed that the page is no longer being used, and the page is replaced. If all pages are still in use, then the page that will not be needed for the longest time is replaced. The time to the next reference is expected to be $t_2 - t_1$.

21.2 ■ XDS-940

The XDS-940 operating system [Lichtenberger and Pirtle 1965] was designed at the University of California at Berkeley. Like the Atlas system,

it used paging for memory management. Unlike the Atlas system, the XDS-940 was a time-shared system.

The paging was used only for relocation; it was not used for demand paging. The virtual memory of any user process was only 16K words, whereas the physical memory was 64K words. Pages were 2K words each. The page table was kept in registers. Since physical memory was larger than virtual memory, several user processes could be in memory at the same time. The number of users could be increased by sharing of pages when the pages contained read-only reentrant code. Processes were kept on a drum and were swapped in and out of memory as necessary.

The XDS-940 system was constructed from a modified XDS-930. The modifications were typical of the changes made to a basic computer to allow an operating system to be written properly. A user-monitor mode was added. Certain instructions, such as I/O and Halt, were defined to be privileged. An attempt to execute a privileged instruction in user mode would trap to the operating system.

A system-call instruction was added to the user-mode instruction set. This instruction was used to create new resources, such as files, allowing the operating system to manage the physical resources. Files, for example, were allocated in 256-word blocks on the drum. A bit map was used to manage free drum blocks. Each file had an index block with pointers to the actual data blocks. Index blocks were chained together.

The XDS-940 system also provided system calls to allow processes to create, start, suspend, and destroy subprocesses. A user programmer could construct a system of processes. Separate processes could share memory for communication and synchronization. Process creation defined a tree structure, where a process is the root and its subprocesses are nodes below it in the tree. Each of the subprocesses could, in turn, create more subprocesses.

21.3 ■ THE

The THE operating system [Dijkstra 1968, McKeag and Wilson 1976 (Chapter 3)] was designed at the Technische Hogeschool at Eindhoven in the Netherlands. It was a batch system running on a Dutch computer, the EL X8, with 32K of 27-bit words. The system was mainly noted for its clean design, particularly its layer structure, and its use of a set of concurrent processes employing semaphores for synchronization.

Unlike the XDS-940 system, however, the set of processes in the THE system was static. The operating system itself was designed as a set of cooperating processes. In addition, five user processes were created, which served as the active agents to compile, execute, and print user programs. When one job was finished, the process would return to the input queue to select another job.

A priority CPU-scheduling algorithm was used. The priorities were recomputed every 2 seconds and were inversely proportional to the amount of CPU time used recently (in the last 8 to 10 seconds). This scheme gave higher priority to I/O-bound processes and to new processes.

Memory management was limited by the lack of hardware support. However, since the system was limited and user programs could be written only in Algol, a software paging scheme was used. The Algol compiler automatically generated calls to system routines, which made sure the requested information was in memory, swapping if necessary. The backing store was a 512K word drum. A 512-word page was used, with an LRU page-replacement strategy.

Another major concern of the THE system was deadlock control. The banker's algorithm was used to provide deadlock avoidance.

Closely related to the THE system is the Venus system [Liskov 1972]. The Venus system was also a layer-structure design, using semaphores to synchronize processes. The lower levels of the design were implemented in microcode, however, providing a much faster system. The memory management was changed to a paged-segmented memory. The system was also designed as a time-sharing system, rather than a batch system.

21.4 ■ RC 4000

The RC 4000 system, like the THE system, was notable primarily for its design concepts. It was designed for the Danish RC 4000 computer by Regenecentralen, particularly by Brinch Hansen [1970, 1973 (Chapter 8)]. The objective was not to design a batch system, or a time-sharing system, or any other specific system. Rather, the goal was to create an operating-system nucleus, or kernel, on which a complete operating system could be built. Thus, the system structure was layered, and only the lower levels — the kernel — were provided.

The kernel supported a collection of concurrent processes. Processes were supported by a round-robin CPU scheduler. Although processes could share memory, the primary communication and synchronization mechanism was the *message system* provided by the kernel. Processes could communicate with each other by exchanging fixed-sized messages of eight words in length. All messages were stored in buffers from a common buffer pool. When a message buffer was no longer required, it was returned to the common pool.

A *message queue* was associated with each process. It contained all the messages that had been sent to that process, but had not yet been received. Messages were removed from the queue in FIFO order. The system supported four primitive operations, which were executed atomically:

- **send-message** (**in** *receiver*, **in** *message*, **out** *buffer*)
- **wait-message** (**out** *sender*, **out** *message*, **out** *buffer*)
- **send-answer** (**out** *result*, **in** *message*, **in** *buffer*)
- **wait-answer** (**out** *result*, **out** *message*, **in** *buffer*)

The last two operations allowed processes to exchange several messages at a time.

These primitives required that a process service its message queue in a FIFO order, and that it block itself while other processes were handling its messages. To remove these restrictions, the developers provided two additional communication primitives. They allowed a process to wait for the arrival of the next message or to answer and service its queue in any order:

- **wait-event** (**in** *previous-buffer*, **out** *next-buffer*, **out** *result*)
- **get-event** (**out** *buffer*)

I/O devices were also treated as processes. The device drivers were code that converted the device interrupts and registers into messages. Thus, a process would write to a terminal by sending that terminal a message. The device driver would receive the message and output the character to the terminal. An input character would interrupt the system and transfer to a device driver. The device driver would create a message from the input character and send it to a waiting process.

21.5 ■ CTSS

The CTSS (Compatible Time-Sharing System) system [Corbato et al. 1962] was designed at MIT as an experimental time-sharing system. It was implemented on an IBM 7090 and eventually supported up to 32 interactive users. The users were provided with a set of interactive commands, which allowed them to manipulate files and to compile and run programs through a terminal.

The 7090 had a 32K memory, made up of 36-bit words. The monitor used 5K words, leaving 27K for the users. User memory images were swapped between memory and a fast drum. CPU scheduling employed a multilevel-feedback-queue algorithm. The time quantum for level i was 2^i time units. If a program did not finish its CPU burst in one time quantum, it was moved down to the next level of the queue, giving it twice as much time. The program at the highest level (with the shortest quantum) was run first. The initial level of a program was determined by its size, so that the time quantum was at least as long as the swap time.

CTSS was extremely successful, and continued to be used as late as 1972. Although it was quite limited, it succeeded in demonstrating that time sharing was a convenient and practical mode of computing. One result of CTSS was increased development of time-sharing systems. Another result was the development of MULTICS.

21.6 ■ MULTICS

The MULTICS operating system [Corbato and Vyssotsky 1965, Organick 1972] was designed at MIT as a natural extension of CTSS. CTSS and other early time-sharing systems were so successful that there was an immediate desire to proceed quickly to bigger and better systems. As larger computers became available, the designers of CTSS set out to create a time-sharing *utility*. Computing service would be provided like electrical power. Large computer systems would be connected by telephone wires to terminals in offices and homes throughout a city. The operating system would be a time-shared system running continuously with a vast file system of shared programs and data.

MULTICS was designed by a team from MIT, GE (which later sold its computer department to Honeywell), and Bell Laboratory (which dropped out of the project in 1969). The basic GE 635 computer was modified to a new computer system called the GE 645, mainly by the addition of paged-segmentation memory hardware.

A virtual address was composed of an 18-bit segment number and a 16-bit word offset. The segments were then paged in 1K word pages. The second-chance page-replacement algorithm was used.

The segmented virtual address space was merged into the file system; each segment was a file. Segments were addressed by the name of the file. The file system itself was a multilevel tree structure, allowing users to create their own subdirectory structures.

Like CTSS, MULTICS used a multilevel feedback queue for CPU scheduling. Protection was accomplished by an access list associated with each file and a set of protection rings for executing processes. The system, which was written almost entirely in PL/1, comprised about 300,000 lines of code. It was extended to a multiprocessor system, allowing a CPU to be taken out of service for maintenance while the system continued running.

21.7 ■ OS/360

The longest line of operating-system development is undoubtedly that for IBM computers. The early IBM computers, such as the IBM 7090 and the IBM 7094, are prime examples of the development of common I/O subroutines, followed by a resident monitor, privileged instructions, memory

protection, and simple batch processing. These systems were developed separately, often by each site independently. As a result, IBM was faced with many different computers, with different languages and different system software.

The IBM/360 was designed to alter this situation. The IBM/360 was designed as a family of computers spanning the complete range from small business machines to large scientific machines. Only one set of software would be needed for these systems, which all used the same operating system: OS/360 [Mealy et al. 1966]. This arrangement was supposed to reduce the maintenance problems for IBM and to allow users to move programs and applications freely from one IBM system to another.

Unfortunately, OS/360 tried to be all things for all people. As a result, it did none of its tasks especially well. The file system included a type field that defined the type of each file, and different file types were defined for fixed-length and variable-length records and for blocked and unblocked files. Contiguous allocation was used, so the user had to guess the size of each output file. The Job Control Language (JCL) added parameters for every possible option, making it incomprehensible to the average user.

The memory-management routines were hampered by the architecture. Although a base-register addressing mode was used, the program could access and modify the base register, so that absolute addresses were generated by the CPU. This arrangement prevented dynamic relocation; the program was bound to physical memory at load time. Two separate versions of the operating system were produced: OS/MFT used fixed regions and OS/MVT used variable regions.

The system was written in assembly language by thousands of programmers, resulting in millions of lines of code. The operating system itself required large amounts of memory for its code and tables. Operating-system overhead often consumed one-half of the total CPU cycles. Over the years, new versions were released to add new features and to fix errors. However, fixing one error often caused another in some remote part of the system, so that the number of known errors in the system was fairly constant.

Virtual memory was added to OS/360 with the change to the IBM 370 architecture. The underlying hardware provided a segmented-paged virtual memory. New versions of OS used this hardware in different ways. OS/VS1 created one large virtual address space, and ran OS/MFT in that virtual memory. Thus, the operating system itself was paged, as well as user programs. OS/VS2 Release 1 ran OS/MVT in virtual memory. Finally, OS/VS2 Release 2, which is now called MVS, provided each user with his own virtual memory.

MVS is still basically a batch operating system. The CTSS system was run on an IBM 7094, but MIT decided that the address space of the 360, IBM's successor to the 7094, was too small for MULTICS, so they switched vendors. IBM then decided to create its own time-sharing system, TSS/360 [Lett and

Konigsford 1968]. Like MULTICS, TSS/360 was supposed to be a large time-shared utility. The basic 360 architecture was modified in the model 67 to provide virtual memory. Several sites purchased the 360/67 in anticipation of TSS/360.

TSS/360 was delayed, however, so other time-sharing systems were developed as temporary systems until TSS/360 was available. A time-sharing option (TSO) was added to OS/360. IBM's Cambridge Scientific Center developed CMS as a single-user system and CP/67 to provide a virtual machine to run it on [Meyer and Seawright 1970, Parmelee et al. 1972].

When TSS/360 was eventually delivered, it was a failure. It was too large and too slow. As a result, no site would switch from its temporary system to TSS/360. Today, time sharing on IBM systems is largely provided either by TSO under MVS or by CMS under CP/67 (renamed VM).

What went wrong with TSS/360 and MULTICS? Part of the problem was that these were advanced systems, and were too large and too complex to be understood. Another problem was the assumption that computing power would be available from a large, remote computer by time sharing. It now appears that most computing will be done by small individual machines — personal computers — not by large, remote time-shared systems that try to be all things to all users.

21.8 ■ Other Systems

There are, of course, other operating systems, and most of them have interesting properties. The MCP operating system for the Burroughs computer family [McKeag and Wilson 1976] was the first to be written in a system programming language. It also supported segmentation and multiple CPUs. The SCOPE operating system for the CDC 6600 [McKeag and Wilson 1976] was also a multi-CPU system. The coordination and synchronization of the multiple processes were surprisingly well designed. Tenex [Bobrow et al. 1972] was an early demand-paging system for the PDP-10, which has had a great influence on subsequent time-sharing systems, such as TOPS-20 for the DEC-20. The VMS operating system for the VAX is based on the RSX operating system for the PDP-11. CP/M was the most common operating system for 8-bit microcomputer systems, few of which exist today; MS-DOS is the most common system for 16-bit microcomputers. Graphical user interfaces, or *GUIs*, are becoming more popular in order to make computers easier to use. The Macintosh Operating System and Microsoft Windows are the two leaders in this area.

APPENDIX

THE NACHOS SYSTEM

By Thomas E. Anderson
University of California, Berkeley

I hear and I forget, I see and I remember,
I do and I understand.

-- Chinese proverb

A good way to gain a deeper understanding of modern operating-system concepts is to get your hands dirty — to take apart the code for an operating system to see how it works at a low level, to build significant pieces of the operating system yourself, and to observe the effects of your work. An operating-system course project provides this opportunity to see how you can use basic concepts to solve real-world problems. Course projects can also be valuable in many other areas of computer science, from compilers and databases to graphics and robotics. But a project is particularly important for operating systems, where many of the concepts are best learned by example and experimentation.

That is why we created *Nachos,* an instructional operating system intended for use as the course project for an undergraduate or first-year graduate course in operating systems. Nachos includes code for a simple but complete working operating system, a machine simulator that allows it to be used in a normal UNIX workstation environment, and a set of sample assignments. Nachos lets anyone explore all the major components of a modern operating system described in this book, from threads and process synchronization, to file systems, to multiprogramming, to virtual memory, to networking. The assignments ask you to design and implement a significant piece of functionality in each of these areas.

Nachos is distributed without charge. It currently runs on both Digital Equipment Corporation MIPS UNIX workstations and Sun SPARC workstations; ports to other machines are in progress. See Section A.4 to learn how to obtain a copy of Nachos.

Here, we give an overview of the Nachos operating system and the machine simulator, and describe our experiences with the example assignments. Of necessity, Nachos is evolving continually, because the field of operating systems is evolving continually. Thus, we can give only a snapshot of Nachos; in Section A.4 we explain how to obtain more up to date information.

A.1 ■ Overview

Many of the earliest operating-system course projects were designed in response to the development of UNIX in the mid-1970s. Earlier operating systems, such as MULTICS and OS/360, were far too complicated for an undergraduate to understand, much less to modify, in one semester.

Even UNIX itself is too complicated for that purpose, but UNIX showed that the core of an operating system can be written in only a few dozen pages, with a few simple but powerful interfaces. However, recent advances in operating systems, hardware architecture, and software engineering have caused many operating-systems projects developed over the past two decades to become out-of-date. Networking and distributed applications are now commonplace. Threads are crucial for the construction of both operating systems and higher-level concurrent applications. And the cost–performance tradeoffs among memory, CPU speed, and secondary storage are now different from those imposed by core memory, discrete logic, magnetic drums, and card readers.

Nachos is intended to help people learn about these modern systems concepts. Nachos illustrates and takes advantage of modern operating-systems technology, such as threads and remote procedure calls; recent hardware advances, such as RISCs and the prevalence of memory hierarchies; and modern software-design techniques, such as protocol layering and object-oriented programming.

In designing Nachos, we faced constantly the tradeoff between simplicity and realism in choosing what code to provide as part of the baseline system, and what to leave for the assignments. We believe that a course project must achieve a careful balance among the time that students spend reading code, that they spend designing and implementing, and that they spend learning new concepts. At one extreme, we could have provided nothing but bare hardware, leaving a *tabula rasa* for students to build an entire operating system from scratch. This approach is

impractical, given the scope of topics to cover. At the other extreme, starting with code that is too realistic would make it easy to lose sight of key ideas in a forest of details.

Our approach was to build the simplest possible implementation for each subsystem of Nachos; this provides a working example — albeit an overly simplistic one — of the operation of each component of an operating system. The baseline Nachos operating-system kernel includes a thread manager, a file system, the ability to run user programs, and a simple network mailbox. As a result of our emphasis on simplicity, the baseline kernel comprises about 2500 lines of code, about one-half of which are devoted to interface descriptions and comments. (The hardware simulator takes up another 2500 lines, but you do not need to understand the details of its operation to do the assignments.) It is thus practical to read, understand, and modify Nachos during a single semester course. By contrast, building a project around a system like UNIX would add realism, but the UNIX 4.3BSD file system *by itself*, even excluding the device drivers, comprises roughly 5000 lines of code. Since a typical course will spend only about 2 to 3 weeks of the semester on file systems, size makes UNIX impractical as a basis for an undergraduate operating-system course project.

We have found that the baseline Nachos kernel can demystify a number of operating-system concepts that are difficult to understand in the abstract. Simply reading and walking through the execution of the baseline system can answer numerous questions about how an operating system works at a low level, such as:

- How do all the pieces of an operating system fit together?
- How does the operating system start a thread? How does it start a process?
- What happens when one thread context switches to another thread?
- How do interrupts interact with the implementation of critical sections?
- What happens on a system call? What happens on a page fault?
- How does address translation work?
- Which data structures in a file system are on disk, and which are in memory?
- What data need to be written to disk when a user creates a file?
- How does the operating system interface with I/O devices?
- What does it mean to build one layer of a network protocol on another?

Of course, reading code by itself can be a boring and pointless exercise; we addressed this problem by keeping the code as simple as possible, and by designing assignments that modify the system in fundamental ways. Because we start with working code, the assignments can focus on the more interesting aspects of operating-system design, where tradeoffs exist and there is no single right answer.

A.2 ■ Nachos Software Structure

Before we discuss the sample assignments in detail, we first outline the structure of the Nachos software. Figure A.1 illustrates how the major pieces in Nachos fit together. Like many earlier instructional operating systems, Nachos runs on a *simulation* of real hardware. Originally, when operating-system projects were first being developed in the 1970s and early 1980s, the reason for using a simulator was to make better use of scarce hardware resources. Without a simulator, each student would need her

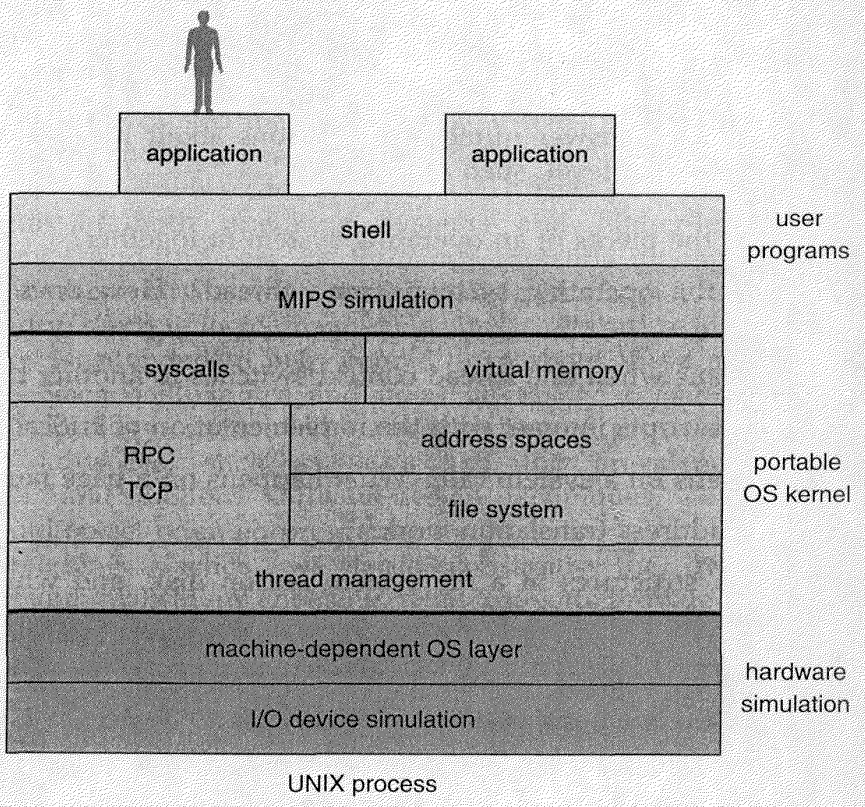

Figure A.1 How the major pieces in Nachos fit together.

own physical machine to test new versions of the kernel. Now that personal computers are commonplace, is there still a reason to develop an operating system on a simulator, rather than on physical hardware?

We believe that the answer is yes, because using a simulator makes debugging easier. On real hardware, operating-system behavior is nondeterministic; depending on the precise timing of interrupts, the operating system may take one path through its code or another. Synchronization can help to make operating-system behavior more predictable, but what if we have a bug in our synchronization code such that two threads can access the same data structure at the same time? The kernel may behave correctly most of the time, yet crash occasionally. Without being able to repeat the behavior that led to the crash, however, it would be difficult to find the root cause of the problem. Running on a simulator, rather than on real hardware, allows us to make system behavior repeatable. Of course, debugging nonrepeatable execution sequences is part of life for professional operating-system engineers, but it did not seem advisable for us to make this experience part of anyone's introduction to operating systems.

Running on simulated hardware has other advantages. During debugging, it is important to be able to make a change to the system quickly, to recompile, and to test the change to see whether it fixed the problem. Using a simulator reduces the time required for this edit–compile–debug cycle, because otherwise the entire system has to be rebooted to test a new version of the kernel. Moreover, normal debugging tools do not work on operating-system kernels, because, for example, if the kernel stops at a breakpoint, the debugger cannot use the kernel to print the prompt for the next debugging command. In practice, debugging an operating-system kernel on real hardware requires *two* machines: one to run the kernel under test, and the other to run the debugger. For these reasons, many commercial operating-system development projects now routinely use simulators to speed development.

One approach would be to simulate the entire workstation hardware, including fetching, decoding, and executing each kernel- or user-mode instruction in turn. Instead, we take a shortcut for performance. The Nachos kernel code executes in native mode as a normal (debuggable) UNIX process linked with the hardware simulator. The simulator surrounds the kernel code, making it appear as though it is running on real hardware. Whenever the kernel code accesses an I/O device — such as a clock chip, a disk, a network controller, or a console — the simulator is invoked to perform the I/O activity. For instance, the simulator implements disk I/O using UNIX file routines; it implements network packet transfer via UNIX sockets.

In addition, we simulate each instruction executed in user mode. Whenever the kernel gives up control to run application code, the simulator fetches each application instruction in turn, checks for page

faults or other exceptions, and then simulates its execution. When an application page fault or hardware interrupt occurs, the simulator passes control back to the kernel for processing, as the hardware would in a real system.

Thus, in Nachos, user applications, the operating-system kernel, and the hardware simulator run together in a normal UNIX process. The UNIX process thus represents a single workstation running Nachos. The Nachos kernel, however, is written as though it were running on real hardware. In fact, we could port the Nachos kernel to a physical machine simply by replacing the hardware simulation with real hardware and a few machine-dependent device-driver routines.

Nachos is different from earlier systems in several significant ways:

1. We can run normal compiled C programs on the Nachos kernel, because we simulate a standard, well-documented, instruction set (MIPS R2/3000 integer instructions) for user-mode programs. In the past, operating-system projects typically simulated their own ad hoc instruction set, requiring user programs to be written in a special-purpose assembly language. However, because the R2/3000 is a RISC, it is straightforward to simulate its instruction set. In all, the MIPS simulation code is only about 10 pages long.

2. We simulate accurately the behavior of a network of workstations, each running a copy of Nachos. We connect Nachos "machines," each running as a UNIX process, via UNIX sockets, simulating a local-area network. A thread on one "machine" can then send a packet to a thread running on a different "machine"; of course, both are simulated on the same physical hardware.

3. The simulation is deterministic, and kernel behavior is reproducible. Instead of using UNIX signals to simulate asynchronous devices such as the disk and the timer, Nachos maintains a simulated time that is incremented whenever a user program executes an instruction and whenever a call is made to certain low-level kernel routines. Interrupt handlers are then invoked when the simulated time reaches the appropriate point. At present, the precise timing of network packet delivery is not reproducible, although this limitation may be fixed in later versions of Nachos.

4. The simulation is randomizable to add unpredictable, but repeatable, behavior to the kernel thread scheduler. Our goal was to make it easy to test kernel behavior given different interleavings of the execution of concurrent threads. Simulated time is incremented whenever interrupts are enabled within the kernel (in other words, whenever any low-level synchronization routine, such as semaphore P or V, is called); after a random interval of simulated time, the scheduler will

cause the current thread to be time sliced. As another example, the network simulation randomly chooses which packets to drop. Provided that the initial seed to the random number generator is the same, however, the behavior of the system is repeatable.

5. We hide the hardware simulation from the rest of Nachos via a machine-dependent interface layer. For example, we define an abstract disk that accepts requests to read and write disk sectors and provides an interrupt handler to be called on request completion. The details of the disk simulator are hidden behind this abstraction, in much the same way that disk-device–specific details are isolated in a normal operating system. One advantage to using a machine-dependent interface layer is to make clear which portions of Nachos can be modified (the kernel and the applications) and which portions are off-limits (the hardware simulation — at least until you take a computer-architecture course).

A.3 ■ Sample Assignments

Nachos contains five major components, each the focus of one assignment given during the semester: thread management and synchronization, the file system, user-level multiprogramming support, the virtual-memory system, and networking. Each assignment is designed to build on previous ones; for instance, every part of Nachos uses thread primitives for managing concurrency. This design reflects part of the charm of developing operating systems: You get to use what you build. It is easy, however, to change the assignments or to do them in a different order.

In Sections A.3.1 through A.3.5, we discuss each of the five assignments in turn, describing what hardware-simulation facilities and operating-system structures we provide, and what we ask you to implement. Of course, because Nachos is continuing to evolve, what is described here is a snapshot of what is available at the time of printing. Section A.4 explains how to obtain more up-to-date information.

The assignments are intended to be of roughly equal size, each taking approximately 3 weeks of a 15-week (semester) course, assuming that two people work together on each. The file-system assignment is the most difficult of the five; the multiprogramming assignment is the least difficult. Faculty who have used Nachos say that they find it useful to spend 1/2 to 1 hour per week discussing the assignments. We have found it useful for faculty to conduct a design review with each pair of students the week before each assignment is due.

Nachos is intended to encourage a quantitative approach to operating-system design. Frequently, the choice of how to implement an operating-system function reduces to a tradeoff between simplicity and performance.

Making informed decisions about tradeoffs is one of the key tasks to learn in an undergraduate operating-system course. The Nachos hardware simulation reflects current hardware performance characteristics (except that kernel execution time is estimated, rather than being measured directly). The assignments exploit this feature by asking that you explain and optimize the performance of your implementations on simple benchmarks.

The Nachos kernel and simulator are implemented in a subset of C++. Object-oriented programming is becoming more popular, and it is a natural idiom for stressing the importance of modularity and clean interfaces in building systems. Unfortunately, C++ is a complicated language; thus, to simplify matters, we omitted certain aspects from standard C++: derived classes, operator and function overloading, C++ streams, and generics. We also kept inlines to a minimum. The Nachos distribution includes a short primer to help people who know C to learn our subset of C++; we have found that our students pick up this subset quickly.

A.3.1 Thread Management

The first assignment introduces the concepts of threads and concurrency. The baseline Nachos kernel provides a basic working thread system and an implementation of semaphores; the assignment is to implement Mesa-style locks and condition variables using semaphores, and then to implement solutions to a number of concurrency problems using these synchronization primitives.

In much the same way as understanding pointers can be difficult for beginning programmers, understanding concurrency requires a conceptual leap. We believe that a good way to learn about concurrency is to take a hands-on approach. Nachos helps to teach concurrency in two ways. First, thread management in Nachos is *explicit*: it is possible to trace, literally statement by statement, what happens during a context switch from one thread to another, from the perspectives of an outside observer and of the threads involved. We believe that this experience is crucial to demystifying concurrency. Precisely because C and C++ allow nothing to be swept under the carpet, concurrency may be easier to understand (although more difficult to use) in these programming languages than in those explicitly designed for concurrency, such as Ada or Modula-3.

Second, a working thread system, like that in Nachos, provides a chance to practice writing, and testing, concurrent programs. Even experienced programmers find it difficult to think concurrently. When we first used Nachos, we omitted many of the practice problems that we now include in the assignment, thinking that students would see enough concurrency in the rest of the project. Later, we realized that many students were still making concurrency errors even in the final phase of the project.

Our primary goal in building the baseline thread system was simplicity, to reduce the effort required to trace through the thread system's behavior. The implementation takes a total of about 10 pages of C++ and one page of MIPS assembly code. For simplicity, thread scheduling is normally nonpreemptive, but to emphasize the importance of critical sections and synchronization, we have a command-line option that causes threads to be time sliced at "random," but repeatable, points in the program. Concurrent programs are correct only if they work when a context switch can happen at any time.

A.3.2 File Systems

Real file systems can be complex artifacts. The UNIX file system, for example, has at least three levels of indirection — the per-process file-descriptor table, the system wide open-file table, and the in-core inode table — before you even get to disk blocks. As a result, to build a file system that is simple enough to read and understand in a couple of weeks, we were forced to make some difficult choices about where to sacrifice realism.

We provide a basic working file system, stripped of as much functionality as possible. Although the file system has an interface similar to that of UNIX (cast in terms of C++ objects), it also has many significant limitations with respect to commercial file systems: there is no synchronization (only one thread at a time can access the file system), files have a very small maximum size, files have a fixed size once created, there is no caching or buffering of file data, the file name space is completely flat (there is no hierarchical directory structure), and there is no attempt to provide robustness across machine and disk crashes. As a result, the basic file system takes only about 15 pages of code.

The assignment is (1) to correct some of these limitations, and (2) to improve the performance of the resulting file system. We list a few possible optimizations, such as caching and disk scheduling, but part of the exercise is to decide which solutions are the most cost effective.

At the hardware level, we provide a disk simulator, which accepts read sector and write sector requests and signals the completion of an operation via an interrupt. The disk data are stored in a UNIX file; read and write sector operations are performed using normal UNIX file reads and writes. After the UNIX file is updated, we calculate how long the simulated disk operation should have taken (from the track and sector of the request), and set an interrupt to occur that far in the future. Read and write sector requests (emulating hardware) return immediately; higher-level software is responsible for waiting until the interrupt occurs.

We made several mistakes in developing the Nachos file system. In our first attempt, the file system was much more realistic than the current one, but it also took more than four times as much code. We were forced

to rewrite it to cut it down to code that could be read and understood quickly. When we handed out this simpler file system, we did not provide sufficient code for it to be working completely; we left out file read and file write to be written as part of the assignment. Although these functions are fairly straightforward to implement, the fact that the code did not work meant that students had difficulty understanding how each of the pieces of the file system fit with the others.

We also initially gave students the option of which limitation to fix; we found that students learned the most from fixing the first four listed. In particular, the students who chose to implement a hierarchical directory structure found that, although it was conceptually simple, the implementation required a relatively large amount of code.

Finally, many modern file systems include some form of write-ahead logging or log structure, simplifying crash recovery. The assignment now completely ignores this issue, but we are currently looking at ways to do crash recovery by adding simple write-ahead logging code to the baseline Nachos file system. As it stands, the choice of whether or not to address crash recovery is simply a tradeoff. In the limited amount of time available, we ask students to focus on how basic file systems work, how the file abstraction allows disk data layout to be changed radically without changing the file-system interface, and how caching can be used to improve I/O performance.

A.3.3 Multiprogramming

In the third assignment, we provide code to create a user address space, to load a Nachos file containing an executable image into user memory, and then to run the program. The initial code is restricted to running only a single user program at a time. The assignment is to expand this base to support multiprogramming, to implement a variety of system calls (such as UNIX **fork** and **exec**) as well as a user-level shell, and to optimize the performance of the resulting system on a mixed workload of I/O- and CPU-bound jobs.

Although we supply little Nachos kernel code as part of this assignment, the hardware simulation does require a fair amount of code. We simulate the entire MIPS R2/3000 integer instruction set and a simple single-level page-table translation scheme. (For this assignment, a program's entire virtual address space must be mapped into physical memory; true virtual memory is left for assignment 4.) In addition, we provide an abstraction that hides most of the details of the MIPS object-code format.

This assignment requires few conceptual leaps, but it does tie together the work of the previous two assignments, resulting in a usable — albeit limited — operating system. Because the simulator can run C programs, it is easy to write utility programs (such as the shell or UNIX **cat**) to exercise

the system. (One overly ambitious student attempted unsuccessfully to port emacs.) The assignment illustrates that there is little difference between writing user code and writing operating-system kernel code, except that user code runs in its own address space, isolating the kernel from user errors.

One important topic that we chose to leave out (again, as a tradeoff against time constraints) is the trend toward a small-kernel operating-system structure, where pieces of the operating system are split off into user-level servers. Because of Nachos' modular design, it would be straightforward to move Nachos toward a small-kernel structure, except that (1) we have no symbolic debugging support for user programs, and (2) we would need a stub compiler to make it easy to make remote procedure calls across address spaces. One reason for adopting a micro-kernel design is that it is easier to develop and debug operating-system code as a user-level server than if the code is part of the kernel. Because Nachos runs as a UNIX process, the reverse is true: It is easier to develop and debug Nachos kernel code than application code running on top of Nachos.

A.3.4 Virtual Memory

Assignment 4 is to replace the simple memory-management system from the previous assignment with a true virtual-memory system — that is, one that presents to each user program the abstraction of an (almost) unlimited virtual-memory size by using main memory as a cache for the disk. We provide no new hardware or operating-system components for this assignment.

The assignment has three parts. The first part is to implement the mechanism for page-fault handling — the kernel must catch the page fault, find the needed page on disk, find a page frame in memory to hold the needed page (writing the old contents of the page frame to disk if the page frame is dirty), read the new page from disk into memory, adjust the page-table entry, and then resume the execution of the program. This mechanism can take advantage of the code written for the previous assignments: The backing store for an address space can be represented simply as a Nachos file, and synchronization is needed when multiple page faults occur concurrently.

The second part of the assignment is to devise a policy for managing the memory as a cache — for deciding which page to toss out when a new page frame is needed, in what circumstances (if any) to do read-ahead, when to write unused dirty pages back to disk, and how many pages to bring in before starting to run a program.

These policy questions can have a large effect on overall system performance, in part because of the large and increasing gap between CPU speed and disk latency — this gap has widened by two orders of magnitude in only the past decade. Unfortunately, the simplest policies

often have unacceptable performance. So that realistic policies are encouraged, the third part of the assignment is to measure the performance of the paging system on a matrix multiply program where the matrices do not fit in memory. This workload is not meant to be representative of real-life paging behavior, but it is simple enough to illustrate the influence of policy changes on application performance. Further, the application illustrates several of the problems with caching: Small changes in the implementation can have a large effect on performance.

A.3.5 Networking

Although distributed systems have become increasingly important commercially, most instructional operating systems do not have a networking component. To address this omission, we chose the capstone of the project to be to write a significant and interesting distributed application.

At the hardware level, each UNIX process running Nachos represents a uniprocessor workstation. We simulate the behavior of a network of workstations by running multiple copies of Nachos, each in its own UNIX process, and by using UNIX sockets to pass network packets from one Nachos "machine" to another. The Nachos operating system can communicate with other systems by sending packets into the simulated network; the transmission is accomplished by socket send and receive. The Nachos network provides unreliable transmission of limited-sized packets from machine to machine. The likelihood that any packet will be dropped can be set as a command-line option, as can the seed used to determine which packets are "randomly" chosen to be dropped. Packets are dropped but are never corrupted, so that checksums are not required.

To show how to use the network and, at the same time, to illustrate the benefits of layering, the Nachos kernel comes with a simple post-office protocol layered on top of the network. The post-office layer provides a set of mailboxes that route incoming packets to the appropriate waiting thread. Messages sent through the post office also contain a return address to be used for acknowledgments.

The assignment is first to provide reliable transmission of arbitrary-sized packets, and then to build a distributed application on top of that service. Supporting arbitrary-sized packets is straightforward — you need merely to split any large packet into fixed-sized pieces, to add fragment serial numbers, and to send the pieces one by one. Ensuring reliability is more interesting, requiring a careful analysis and design. To reduce the time required to do the assignment, we do not ask you to implement congestion control or window management, although of course these are important issues in protocol design.

The choice of how to complete the project is left open. We do make a few suggestions: multiuser UNIX talk, a distributed file system with caching, a process-migration facility, distributed virtual memory, a gateway protocol that is robust to machine crashes. Perhaps the most interesting application that a student built (that we know of) was a distributed version of the "battleship" game, with each player on a different machine. This application illustrated the role of distributed state, since each machine kept only its local view of the gameboard; it also exposed several performance problems in the hardware simulation, which we have since fixed.

Perhaps the biggest limitation of the current implementation is that we do not model network performance correctly, because we do not keep the timers on each of the Nachos machines synchronized with one another. We are currently working on fixing this problem, using distributed simulation techniques for efficiency. These techniques will allow us to make performance comparisons between alternate implementations of network protocols; they will also enable us to use the Nachos network as a simulation of a message-passing multiprocessor.

A.4 ■ Information on Obtaining a Copy of Nachos

You can obtain Nachos by anonymous ftp from the machine ftp.cs.berkeley.edu by following these steps:

1. Use UNIX ftp to access ftp.cs.berkeley.edu:

   ftp ftp.cs.berkeley.edu

2. You will get a login prompt. Type the word anonymous, and then use your e-mail address as the password.

   Name: anonymous
   Password: tea@cs.berkeley.edu (for example)

3. You are now in ftp. Move to the Nachos subdirectory.

   ftp> cd ucb/nachos

4. You must remember to turn on "binary" mode in ftp; unfortunately, if you forget to do so, when you fetch the Nachos file, it will be garbled without any kind of warning message. This error is one of the most common that people make in obtaining software using anonymous ftp.

   ftp> binary

5. You can now copy the compressed UNIX tar file containing the Nachos distribution to your machine. The software will automatically enroll you in a mailing list for announcements of new releases of Nachos; you can remove yourself from this list by sending e-mail to nachos@cs.berkeley.edu.

```
ftp> get nachos.tar.Z
```

6. Exit the ftp program:

```
ftp> quit
```

7. Decompress and detar to obtain the Nachos distribution. (If the decompress step fails, you probably forgot to set binary mode in ftp in step 4. You will need to start over.)

```
uncompress nachos.tar.Z
tar -xf nachos.tar
```

8. The preceding steps will produce several files, including the code for the baseline Nachos kernel, the hardware simulator, documentation on the sample assignments, and the C++ primer. There will also be a README file to get you started: It explains how to build the baseline system, how to print out documentation, and which machine architectures are currently supported.

```
cat README
```

Mendel Rosenblum at Stanford has ported the Nachos kernel to run on Sun SPARC workstations, although user programs running on top of Nachos must still be compiled for the MIPS R2/3000 RISC processor. Ports to machines other than Digital Equipment Corporation MIPS UNIX workstations and Sun SPARC workstations are in progress. Up-to-date information on machine availability is included in the README file in the distribution. The machine dependence comes in two parts. First, the Nachos kernel runs just like normal application code on a UNIX workstation, but a small amount of assembly code is needed in the Nachos kernel to implement thread context switching. Second, Nachos simulates the instruction-by-instruction execution of user programs, to catch page faults and other exceptions. This simulation assumes the MIPS R2/3000 instruction set. To port Nachos to a new machine, we replace the kernel thread-switch code with machine-specific code, and rely on a C cross-compiler to generate MIPS object code for each user program. (A cross-compiler is a compiler that generates object code for one machine type while running on a different

machine type.) Because we rely on a cross-compiler, we do not have to rewrite the instruction-set simulator for each port to a new machine. The SPARC version of Nachos, for instance, comes with instructions on how to cross-compile to MIPS on the SPARC.

Questions about Nachos and bug reports should be directed via e-mail to nachos@cs.berkeley.edu. Questions can also be posted to the alt.os.nachos newsgroup.

A.5 ■ Conclusions

Nachos is an instructional operating system designed to reflect recent advances in hardware and software technology, to illustrate modern operating-system concepts, and, more broadly, to help teach the design of complex computer systems. The Nachos kernel and sample assignments illustrate principles of computer-system design needed to understand the computer systems of today and of the future: concurrency and synchronization, caching and locality, the tradeoff between simplicity and performance, building reliability from unreliable components, dynamic scheduling, object-oriented programming, the power of a level of translation, protocol layering, and distributed computing. Familiarity with these concepts is valuable, we believe, even for those people who do not end up working in operating-system development.

■ Bibliographic Notes

Wayne Christopher, Steve Procter, and Thomas Anderson (the author of this appendix) did the initial implementation of Nachos in January 1992. The first version was used for one term as the project for the undergraduate operating-systems course at The University of California at Berkeley. We then revised both the code and the assignments, releasing Nachos, Version 2 for public distribution in August 1992; Mendel Rosenblum ported Nachos to the Sun SPARC workstation. The second version is currently in use at several universities including Carnegie Mellon, Colorado State, Duke, Harvard, Stanford, State University of New York at Albany, University of Washington, and, of course, Berkeley; we have benefited tremendously from the suggestions and criticisms of our early users.

In designing the Nachos project, we have borrowed liberally from ideas found in other systems, including the TOY operating system project, originally developed by Ken Thompson while he was at Berkeley, and modified extensively by a collection of people since then; Tunis, developed by Rick Holt [Holt 1983]; and Minix, developed by Andy Tanenbaum

[Tanenbaum 1987]. Lions [1977] was one of the first people to realize that the core of an operating system could be expressed in a few lines of code, and then used to teach people about operating systems. The instruction-set simulator used in Nachos is largely based on a MIPS simulator written by John Ousterhout.

We credit Lance Berc with inventing the acronym "Nachos" Not Another Completely Heuristic Operating System.

BIBLIOGRAPHY

[Abbot 1984] C. Abbot, "Intervention Schedules for Real-Time Programming," *IEEE Transactions on Software Engineering,* Volume SE-10, Number 3 (May 1984), pages 268–274.

[Accetta et al. 1986] M. Accetta, R. Baron, W. Bolosky, D. B. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: A New Kernel Foundation for Unix Development," *Proceedings of the Summer 1986 USENIX Conference* (June 1986), pages 93–112.

[Agrawal et al. 1986] D. P. Agrawal, V. K. Janakiram, and G. C. Pathak, "Evaluating the Performance of Multicomputer Configurations," *Communications of the ACM,* Volume 29, Number 5 (May 1986), pages 23–37.

[Agrawal and El Abbadi 1991] D. P. Agrawal and A. El Abbadi, "An Efficient and Fault-Tolerant Solution of Distributed Mutual Exclusion," *ACM Transactions on Computer Systems,* Volume 9, Number 1 (February 1991), pages 1–20.

[Ahituv et al. 1987] N. Ahituv, Y. Lapid and S. Neumann, "Processing Encrypted Data," *Communications of the ACM,* Volume 30, Number 9 (September 1987), pages 777–780.

[Akl 1983] S. G. Akl, "Digital Signatures: A Tutorial Survey," *Computer,* Volume 16, Number 2 (February 1983), pages 15–24.

[Ammon et al. 1985] G. J. Ammon, J. A. Calabria, and D. T. Thomas, "A High-Speed, Large-Capacity, 'Jukebox' Optical Disk System," *Computer,* Volume 18, Number 7 (July 1985), pages 36–48.

[Anderson et al. 1989] T. E. Anderson, E. D. Lazowska, and H. M. Levy, "The Performance Implications of Thread Management Alternatives for Shared-Memory Multiprocessors," *IEEE Transactions on Computers,* Volume 38, Number 12 (December 1989), pages 1631–1644.

[Anderson et al. 1991] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy, "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism," *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles* (October 1991), pages 95–109.

[Anyanwu and Marshall 1986] J. A. Anyanwu and L. F. Marshall, "A Crash Resistant UNIX File System," *Software—Practice and Experience,* Volume 16 (February 1986), pages 107–118.

[Apple 1987] Apple Computer Inc., *Apple Technical Introduction to the Macintosh Family,* Addison-Wesley, Reading, MA (1987).

[Apple 1991] Apple Computer Inc., *Inside Macintosh, Volume VI,* Addison-Wesley, Reading, MA (1991).

[Artsy 1989a] Y. Artsy, Ed., Special Issue on Process Migration. *Newsletter of the IEEE Computer Society Technical Committee on Operating Systems* (Winter 1989).

[Artsy 1989b] Y. Artsy, "Designing a Process Migration Facility: The Charlotte Experience," *Computer,* Volume 22, Number 9 (September 1989), pages 47–56.

[AT&T 1986] AT&T, Steven V. Earhart, Ed., *UNIX Programmer's Manual,* Holt, Rinehart, and Winston, New York, NY (1986).

[Babaoglu and Joy 1981] O. Babaoglu and W. Joy, "Converting a Swap-Based System to Do Paging in an Architecture Lacking Page-Referenced Bits," *Proceedings of the Eighth ACM Symposium on Operating System Principles* (December 1981), pages 78–86.

[Bach 1987] M. J. Bach, *The Design of the UNIX Operating System,* Prentice-Hall, Englewood Cliffs, NJ (1987).

[Baer 1980] J. L. Baer, *Computer System Architecture,* Computer Science Press, Rockville, MD (1980).

[Balkovich et al. 1985] E. Balkovich, S. R. Lerman, and R. P. Parmelee, "Computing in Higher Education: The Athena Experience," *Communications of the ACM,* Volume 28, Number 11 (November 1985), pages 1214–1224.

[Barak and Kornatzky 1987] A. Barak and Y. Kornatzky, "Design Principles of Operating Systems for Large Scale Multicomputers," *Experience with Distributed Systems, Lecture Notes in Computer Science,* Volume 309, Springer-Verlag (September 1987), pages 104–123.

[Bayer et al. 1978] R. Bayer, R. M. Graham, and G. Seegmuller, Eds., *Operating Systems — An Advanced Course,* Springer-Verlag, Berlin (1978).

[Bays 1977] C. Bays, "A Comparison of Next-Fit, First-Fit, and Best-Fit," *Communications of the ACM,* Volume 20, Number 3 (March 1977), pages 191–192.

[Belady 1966] L. A. Belady, "A Study of Replacement Algorithms for a Virtual-Storage Computer," *IBM Systems Journal,* Volume 5, Number 2 (1966), pages 78–101.

[Belady et al. 1969] L. A. Belady, R. A. Nelson, and G. S. Shedler, "An Anomaly in Space-Time Characteristics of Certain Programs Running in a Paging Machine," *Communications of the ACM,* Volume 12, Number 6 (June 1969), pages 349–353.

[Ben-Ari 1990] M. Ben-Ari, *Principles of Concurrent and Distributed Programming,* Prentice-Hall, Englewood Cliffs, NJ (1990).

[Bernstein and Goodman 1980] P. A. Bernstein and N. Goodman, "Timestamp-based Algorithms for Concurrency Control in Distributed Database Systems," *Proceedings of the International Conference on Very Large Data Bases* (1980), pages 285–300.

[Bernstein and Siegel 1975] A. J. Bernstein and P. Siegel, "A Computer Architecture for Level Structured Operating Systems," *IEEE Transactions on Computers,* Volume C-24, Number 8 (August 1975), pages 785–793.

[Bernstein et al. 1987] A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems,* Addison-Wesley, Reading, MA (1987).

[Bershad and Pinkerton 1988] B. N. Bershad and C. B. Pinkerton, "Watchdogs: Extending the Unix File System," *Proceedings of the Winter 1988 USENIX Conference* (February 1988).

[Bershad et al. 1990] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy, "Lightweight Remote Procedure Call," *ACM Transactions on Computer Systems,* Volume 8, Number 1 (February 1990), pages 37–55.

[Bhuyan et al. 1989] L. N. Bhuyan, Q. Yang, and D. P. Agrawal, "Performance of Multiprocessor Interconnection Networks," *Computer,* Volume 22, Number 2 (February 1989), pages 25–37.

[Bic and Shaw 1988] L. Bic and A. C. Shaw, *The Logical Design of Operating Systems,* Second Edition, Prentice-Hall, Englewood Cliffs, NJ (1988).

[Birman and Joseph 1987] K. Birman and T. Joseph, "Reliable Communication in the Presence of Failures," *ACM Transactions on Computer Systems,* Volume 5, Number 1 (February 1987).

[Birrell 1989] A. D. Birrell, "An Introduction to Programming with Threads," Technical Report 35, DEC-SRC, Palo Alto, CA (January 1989).

[Birrell and Nelson 1984] A. D. Birrell and B. J. Nelson, "Implementing Remote Procedure Calls," *ACM Transactions on Computer Systems,* Volume 2, Number 1 (February 1984), pages 39–59.

[Black 1990] D. L. Black, "Scheduling Support for Concurrency and Parallelism in the Mach Operating System," *IEEE Computer* (May 1990), pages 35–43.

[Black et al. 1988] D. L. Black, D. B. Golub, R. F. Rashid, A. Tevanian Jr., and M. Young, "The Mach Exception Handling Facility," Technical Report, Carnegie-Mellon University (April 1988).

[Black et al. 1992] D. L. Black, D. B. Golub, D. P. Julin, R. F. Rashid, R. P. Draves, R. W. Dean, A. Forin, J. Barrera, H. Tokadu, G. Malan, and D. Bohman, "Microkernel Operating System Architecture and Mach," *Proceedings of the USENIX Workshop on Micro-Kernels and Other Kernel Architectures* (April 1992), pages 11–30.

[Blair et al. 1985] G. S. Blair, J. R. Malone, and J. A. Mariani, "A Critique of UNIX," *Software—Practice and Experience,* Volume 15, Number 6 (December 1985), pages 1125–1139.

[Bobrow et al. 1972] D. G. Bobrow, J. D. Burchfiel, D. L. Murphy, and R. S. Tomlinson, "TENEX, a Paged Time Sharing System for the PDP-10," *Communications of the ACM,* Volume 15, Number 3 (March 1972).

[Boorstyn and Frank 1977] R. R. Boorstyn and H. Frank, "Large-Scale Network Topological Optimization," *IEEE Transactions on Communications,* Volume COM-25, Number 1 (January 1977), pages 29–47.

[Bourne 1978] S. R. Bourne, "The UNIX Shell," *Bell System Technical Journal,* Volume 57, Number 6 (July-August 1978), pages 1971–1990.

[Bourne 1983] S. R. Bourne, *The UNIX System,* Addison-Wesley, Reading, MA (1983).

[Boykin et al. 1993] J. Boykin, D. Kirschen, A. Langerman, and S. LoVerso, *Programming under Mach,* Addison-Wesley, Reading, MA (1993).

[Boykin and Langerman 1990] J. Boykin and A. B. Langerman, "Mach/4.3BSD: A Conservative Approach to Parallelization," *Computing Systems,* Volume 3, Number 1 (Winter 1990).

[Brent 1989] R. Brent, "Efficient Implementation of the First-Fit Strategy for Dynamic Storage Allocation," *ACM Transactions on Programming Languages and Systems* (July 1989).

[Brereton 1986] O. P. Brereton, "Management of Replicated Files in a UNIX Environment," *Software—Practice and Experience,* Volume 16 (August 1986), pages 771–780.

[Brinch Hansen 1970] P. Brinch Hansen, "The Nucleus of a Multiprogramming System," *Communications of the ACM,* Volume 13, Number 4 (April 1970), pages 238–241 and 250.

[Brinch Hansen 1972] P. Brinch Hansen, "Structured Multiprogramming," *Communications of the ACM,* Volume 15, Number 7 (July 1972), pages 574–578.

[Brinch Hansen 1973] P. Brinch Hansen, *Operating System Principles,* Prentice-Hall, Englewood Cliffs, NJ (1973).

[Brownbridge et al. 1982] D. R. Brownbridge, L. F. Marshall, and B. Randell, "The Newcastle Connection or UNIXes of the World Unite!" *Software—Practice and Experience,* Volume 12, Number 12 (December 1982), pages 1147–1162.

[Brumfield 1986] J. A. Brumfield, "A Guide to Operating Systems Literature," *Operating Systems Review,* Volume 20, Number 2 (April 1986), pages 38–42.

[Brunt and Tuffs 1976] R. F. Brunt and D. E. Tuffs, "A User-Oriented Approach to Control Languages," *Software—Practice and Experience,* Volume 6, Number 1 (January-March 1976), pages 93–108.

[BSTJ 1978] "UNIX Time-Sharing System," *The Bell System Technical Journal,* Volume 57, Number 6, Part 2 (July-August 1978).

[BSTJ 1984] "The UNIX System," *The Bell System Technical Journal,* Volume 63, Number 8, Part 2 (October 1984).

[Burns 1978] J. E. Burns, "Mutual Exclusion with Linear Waiting Using Binary Shared Variables," *SIGACT News,* Volume 10, Number 2 (Summer 1978), pages 42–47.

[Carvalho and Roucairol 1983] O. S. Carvalho and G. Roucairol, "On Mutual Exclusion in Computer Networks," *Communications of the ACM,* Volume 26, Number 2 (February 1983), pages 146–147.

[Carr and Hennessy 1981] W. R. Carr and J. L. Hennessy, "WSClock — A Simple and Effective Algorithm for Virtual Memory Management," *Proceedings of the Eighth Symposium on Operating System Principles* (December 1981), pages 87–95.

[Caswell and Black 1989] D. Caswell and D. Black, "Implementing a Mach Debugger for Multithreaded Applications," Technical Report, Carnegie-Mellon University, PA (November 1989).

[Caswell and Black 1990] D. Caswell and D. Black, "Implementing a Mach Debugger for Multithreaded Applications," *Proceedings of the Winter 1990 USENIX Conference* (January 1990), pages 25–40.

[Cerf and Cain 1983] V. G. Cerf and E. Cain, "The DoD Internet Architecture Model," *Computer Networks,* Volume 7, Number 5 (October 1983), pages 307–318.

[Chang 1980] E. Chang, "N-Philosophers: An Exercise in Distributed Control," *Computer Networks,* Volume 4, Number 2 (April 1980), pages 71–76.

[Chang and Mergen 1988] A. Chang and M. F. Mergen, "801 Storage: Architecture and Programming," *ACM Transactions on Computer Systems,* Volume 6, Number 1 (February 1988), pages 28–50.

[Chen and Patterson 1990] P. Chen and D. Patterson, "Maximizing Performance in a Striped Disk Array," *Proceedings of the Seventeenth Annual International Symposium on Computer Architecture* (May 1990).

[Cheriton and Zwaenepoel 1983] D. R. Cheriton and W. Z. Zwaenepoel, "The Distributed V Kernel and Its Performance for Diskless Workstations," *Proceedings of the Ninth Symposium on Operating Systems Principles* (October 1983), pages 129–140.

[Cheriton et al. 1979] D. R. Cheriton, M. A. Malcolm, L. S. Melen, and G. R. Sager, "Thoth, a Portable Real-Time Operating System," *Communications of the ACM,* Volume 22, Number 2 (February 1979), pages 105–115.

[Chi 1982] C. S. Chi, "Advances in Computer Mass Storage Technology," *Computer,* Volume 15, Number 5 (May 1982), pages 60–74.

[Chow and Abraham 1982] T. C. K. Chow and J. A. Abraham, "Load Balancing in Distributed Systems," *IEEE Transactions on Software Engineering,* Volume SE-8, Number 4 (July 1982), pages 401–412.

[Chu and Opderbeck 1976] W. W. Chu and H. Opderbeck, "Program Behavior and the Page-Fault-Frequency Replacement Algorithm," *Computer,* Volume 9, Number 11 (November 1976), pages 29–38.

[Coffman and Kleinrock 1968] E. G. Coffman and L. Kleinrock, "Feedback Queuing Models for Time-Shared Systems," *Journal of the ACM,* Volume 15, Number 4 (October 1968), pages 549–576.

[Coffman and Denning 1973] E. G. Coffman and P. J. Denning, *Operating Systems Theory,* Prentice-Hall, Englewood Cliffs, NJ (1973).

[Coffman et al. 1971] E. G. Coffman, M. J. Elphick, and A. Shoshani, "System Deadlocks," *Computing Surveys,* Volume 3, Number 2 (June 1971), pages 67–78.

[Cohen and Jefferson 1975] E. S. Cohen and D. Jefferson, "Protection in the Hydra Operating System," *Proceedings of the Fifth Symposium on Operating System Principles* (November 1975), pages 141–160.

[Comer 1984] D. Comer, *Operating System Design: the Xinu Approach,* Prentice-Hall, Englewood Cliffs, NJ (1984).

[Comer 1987] D. Comer, *Operating System Design — Volume II: Internetworking with Xinu,* Prentice-Hall, Englewood Cliffs, NJ (1987).

[Comer 1991] D. Comer, *Internetworking with TCP/IP, Volume I,* Second Edition, Prentice-Hall, Englewood Cliffs, NJ (1991).

[Comer and Stevens 1991] D. Comer and D. L. Stevens, *Internetworking with TCP/IP, Volume II,* Prentice-Hall, Englewood Cliffs, NJ (1991).

[Comer and Stevens 1993] D. Comer and D. L. Stevens, *Internetworking with TCP/IP Principles, Volume III,* Prentice-Hall, Englewood Cliffs, NJ (1993).

[Cooper and Draves 1987] E. C. Cooper and R. P. Draves, "C Threads," Technical Report, Carnegie-Mellon University, PA (July 1987).

[Corbato and Vyssotsky 1965] F. J. Corbato and V. A. Vyssotsky, "Introduction and Overview of the MULTICS System," *Proceedings of the AFIPS Fall Joint Computer Conference* (1965), pages 185–196.

[Corbato et al. 1962] F. J. Corbato, M. Merwin-Daggett, and R. C. Daley, "An Experimental Time-Sharing System," *Proceedings of the AFIPS Fall Joint Computer Conference* (1962), pages 335–344.

[Courtois et al. 1971] P. J. Courtois, F. Heymans, and D. L. Parnas, "Concurrent Control with 'Readers' and 'Writers'," *Communications of the ACM,* Volume 14, Number 10 (October 1971), pages 667–668.

[Creasy 1981] R. J. Creasy, "The Origin of the VM/370 Time-Sharing System," *IBM Journal of Research and Development,* Volume 25, Number 5 (September 1981), pages 483–490.

[CSRG 1986] Computer Systems Research Group — University of California at Berkeley, *BSD UNIX Reference Manuals,* six volumes, USENIX Association (1986).

[Custer 1993] H. Custer, *Inside Windows/NT,* Microsoft Press, Redmond, WA (1993).

[Davcev and Burkhard 1985] D. Davcev and W. A. Burkhard, "Consistency and Recovery Control for Replicated Files," *Proceedings of the Tenth Symposium on Operating Systems Principles,* Volume 19, Number 5 (December 1985), pages 87–96.

[Davies 1983] D. W. Davies, "Applying the RSA Digital Signature to Electronic Mail," *Computer,* Volume 16, Number 2 (February 1983), pages 55–62.

[Day and Zimmerman 1983] J. D. Day and H. Zimmerman, "The OSI Reference Model," *Proceedings of the IEEE,* Volume 71 (December 1983), pages 1334–1340.

[deBruijn 1967] N. G. deBruijn, "Additional Comments on a Problem in Concurrent Programming and Control," *Communications of the ACM,* Volume 10, Number 3 (March 1967), pages 137–138.

[Deitel 1990] H. M. Deitel, *An Introduction to Operating Systems,* Second Edition, Addison-Wesley, Reading, MA (1990).

[Deitel and Kogan 1992] H. M. Deitel and M. S. Kogan, *The Design of OS/2,* Addison-Wesley, Reading, MA (1992).

[Denning 1968] P. J. Denning, "The Working Set Model for Program Behavior," *Communications of the ACM,* Volume 11, Number 5 (May 1968), pages 323–333.

[Denning 1971] P. J. Denning, "Third Generation Computer System," *Computing Surveys,* Volume 3, Number 34 (December 1971), pages 175–216.

[Denning 1980] P. J. Denning, "Working Sets Past and Present," *IEEE Transactions on Software Engineering,* Volume SE-6, Number 1 (January 1980), pages 64–84.

[Denning 1982] D. E. Denning, *Cryptography and Data Security,* Addison-Wesley, Reading, MA (1982).

[Denning 1983] D. E. Denning, "Protecting Public Keys and Signature Keys," *IEEE Computer,* Volume 16, Number 2 (February 1983), pages 27–35.

[Denning 1984] D. E. Denning, "Digital Signatures with RSA and Other Public-Key Cryptosystems," *Communications of the ACM,* Volume 27, Number 4 (April 1984), pages 388–392.

[Dennis 1965] J. B. Dennis, "Segmentation and the Design of Multiprogrammed Computer Systems," *Journal of the ACM,* Volume 12, Number 4 (October 1965), pages 589–602.

[Dennis and Van Horn 1966] J. B. Dennis and E. C. Van Horn, "Programming Semantics for Multiprogrammed Computations," *Communications of the ACM,* Volume 9, Number 3 (March 1966), pages 143–155.

[Diffie and Hellman 1976] W. Diffie and M. E. Hellman, "New Directions in Cryptography," *IEEE Transactions on Information Theory,* Volume 22, Number 6 (November 1976), pages 644–654.

[Diffie and Hellman 1979] W. Diffie and M. E. Hellman, "Privacy and Authentication," *Proceedings of the IEEE,* Volume 67, Number 3 (March 1979), pages 397–427.

[Digital 1981] Digital Equipment Corporation, *VAX Architecture Handbook,* Digital Equipment Corporation, Maynard, MA (1981).

[Dijkstra 1965a] E. W. Dijkstra, "Cooperating Sequential Processes," Technical Report EWD-123, Technological University, Eindhoven, the Netherlands (1965); reprinted in [Genuys 1968], pages 43–112.

[Dijkstra 1965b] E. W. Dijkstra, "Solution of a Problem in Concurrent Programming Control," *Communications of the ACM,* Volume 8, Number 9 (September 1965), Page 569.

[Dijkstra 1968] E. W. Dijkstra, "The Structure of the THE Multiprogramming System," *Communications of the ACM,* Volume 11, Number 5 (May 1968), pages 341–346.

[Dijkstra 1971] E. W. Dijkstra, "Hierarchical Ordering of Sequential Processes," *Acta Informatica,* Volume 1, Number 2 (1971), pages 115–138; reprinted in [Hoare and Perrott 1972], pages 72–93.

[Doeppner 1987] T. W. Doeppner, "Threads: A System for the Support of Concurrent Programming," Technical Report CS-87-11, Department of Computer Science, Brown University (June 1987).

[Donnelley 1979] J. E. Donnelley, "Components of a Network-Operating System," *Computer Networks,* Volume 3, Number 6 (December 1979), pages 389–399.

[Douglis and Ousterhout 1987] F. Douglis and J. Ousterhout, "Process Migration in the Sprite Operating System," *Proceedings of the Seventh IEEE International Conference on Distributed Computing Systems* (1987), pages 18–25.

[Douglis and Ousterhout 1989] F. Douglis and J. Ousterhout, "Process Migration in Sprite: A Status Report," *Newsletter of the IEEE Computer Society Technical Committee on Operating Systems* (Winter 1989).

[Douglis et al. 1991] F. Douglis, M. F. Kaashoek, and A. S. Tanenbaum, "A Comparison of Two Distributed Systems: Amoeba and Sprite," *Computing Systems,* Volume 4 (Fall 1991).

[Draves et al. 1989] R. P. Draves, M. B. Jones, and M. R. Thompson, "MIG — The MACH Interface Generator," Technical Report, Carnegie-Mellon University, PA (November 1989).

[Draves et al. 1991] R. P. Draves, B. N. Bershad, R. F. Rashid, and R. W. Dean, "Using Continuations to Implement Thread Management and Communication in Operating Systems," *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles* (October 1991), pages 122–136.

[Duncan 1990] R. Duncan, "A Survey of Parallel Computer Architectures," *IEEE Computer,* Volume 23, Number 2 (February 1990), pages 5–16.

[Eager et al. 1986] D. Eager, E. Lazowska, and J. Zahorjan, "Adaptive Load Sharing in Homogeneous Distributed Systems," *IEEE Transactions on Software Engineering,* Volume SE-12, Number 5 (May 1986), pages 662–675.

[Eisenberg and McGuire 1972] M. A. Eisenberg and M. R. McGuire, "Further Comments on Dijkstra's Concurrent Programming Control Problem," *Communications of the ACM,* Volume 15, Number 11 (November 1972), page 999.

[Ekanadham and Bernstein 1979] K. Ekanadham and A. J. Bernstein, "Conditional Capabilities," *IEEE Transactions on Software Engineering,* Volume SE-5, Number 5 (September 1979), pages 458–464.

[English and Stepanov 1992] R. M. English and A. A. Stepanov, "Loge: A Self-Organizing Disk Controller," *Proceedings of the Winter 1992 USENIX Conference,* San Francisco, CA (January 1992), pages 238–252.

[Eskicioglu 1990] M. Eskicioglu, "Design Issues of Process Migration Facilities in Distributed Systems," *Newsletter of the IEEE Computer Society Technical Committee on Operating Systems* (Summer 1990).

[Eswaran et al. 1976] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, "The Notions of Consistency and Predicate Locks in a Database System," *Communications of the ACM,* Volume 19, Number 11 (November 1976), pages 624–633.

[Eykholt et al. 1992] J. R. Eykholt, S. R. Kleiman, S. Barton, S. Faulkner, A. Shivalingiah, M. Smith, D. Stein, J. Voll, M. Weeks, and D. Williams, "Beyond Multiprocessing: Multithreading the SunOS Kernel," *Proceedings of the Summer 1992 USENIX Conference* (June 1992), pages 11–18.

[Farrow 1986a] R. Farrow, "Security Issues and Strategies for Users," *UNIX World* (April 1986), pages 65–71.

[Farrow 1986b] R. Farrow, "Security for Superusers, or How to Break the UNIX System," *UNIX World* (May 1986), pages 65–70.

[Feitelson and Rudolph 1990] D. Feitelson and L. Rudolph, "Mapping and Scheduling in a Shared Parallel Environment Using Distributed Hierarchical Control," *Proceedings of the 1990 International Conference on Parallel Processing* (August 1990).

[Ferguson et al. 1988] D. Ferguson, Y. Yemini, and C. Nikolaou, "Microeconomic Algorithms for Load Balancing in Distributed Computer Systems," *Proceedings of the Eighth IEEE International Conference on Distributed Computing Systems* (1988), pages 491–499.

[Feng 1981] T. Feng, "A Survey of Interconnection Networks," *Computer,* Volume 14, Number 12 (1981), pages 12–27.

[Filipski and Hanko 1986] A. Filipski and J. Hanko, "Making UNIX Secure," *Byte* (April 1986), pages 113–128.

[Finkel 1988] R. A. Finkel, *Operating Systems Vade Mecum,* Second Edition, Prentice-Hall, Englewood Cliffs, NJ (1988).

[Folk and Zoellick 1987] M. J. Folk and B. Zoellick, *File Structures,* Addison-Wesley, Reading, MA (1987).

[Forsdick et al. 1978] H. C. Forsdick, R. E. Schantz, and R. H. Thomas, "Operating Systems for Computer Networks," *Computer,* Volume 11, Number 1 (January 1978), pages 48–57.

[Fortier 1989] P. J. Fortier, *Handbook of LAN Technology,* McGraw Hill, New York, NY (1989).

[Frank 1976] G. R. Frank, "Job Control in the MU5 Operating System," *Computer Journal,* Volume 19, Number 2 (May 1976), pages 139–143.

[Freedman 1983] D. H. Freedman, "Searching for Denser Disks," *Infosystems* (September 1983), page 56.

[Fujitani 1984] L. Fujitani, "Laser Optical Disk: The Coming Revolution in On-Line Storage," *Communications of the ACM,* Volume 27, Number 6 (June 1984), pages 546–554.

[Gait 1988] J. Gait, "The Optical File Cabinet: A Random-Access File System for Write-On Optical Disks," *Computer,* Volume 21, Number 6 (June 1988).

[Garcia-Molina 1982] H. Garcia-Molina, "Elections in Distributed Computing Systems," *IEEE Transactions on Computers,* Volume C-31, Number 1 (January 1982).

[Garfinkel and Spafford 1991] S. Garfinkel and G. Spafford, *Practical UNIX Security,* O'Reilly & Associates, Inc., Sebastopol, CA (1991).

[Geist and Daniel 1987] R. Geist and S. Daniel, "A Continuum of Disk Scheduling Algorithms," *ACM Transactions on Computer Systems,* Volume 5, Number 1 (February 1987), pages 77–92.

[Genuys 1968] F. Genuys (Editor), *Programming Languages*, Academic Press, London, England (1968).

[Gerla and Kleinrock 1977] M. Gerla and L. Kleinrock, "Topological Design of Distributed Computer Networks," *IEEE Transactions on Communications,* Volume COM-25, Number 1 (January 1977), pages 48–60.

[Gifford 1982] D. K. Gifford, "Cryptographic Sealing for Information Secrecy and Authentication," *Communications of the ACM,* Volume 25, Number 4 (April 1982), pages 274–286.

[Golden and Pechura 1986] D. Golden and M. Pechura, "The Structure of Microcomputer File Systems," *Communications of the ACM,* Volume 29, Number 3 (March 1986), pages 222–230.

[Goldman 1989] P. Goldman, " Mac VM Revealed," *Byte* (September 1989).

[Grampp and Morris 1984] F. T. Grampp and R. H. Morris, "UNIX Operating-System Security," *AT&T Bell Laboratories Technical Journal,* Volume 63 (October 1984), pages 1649–1672.

[Gray 1978] J. N. Gray, "Notes on Data Base Operating Systems," in [Bayer et al. 1978], pages 393–481.

[Gray 1981] J. N. Gray, "The Transaction Concept: Virtues and Limitations," *Proceedings of the International Conference on Very Large Data Bases* (1981), pages 144–154.

[Gray et al. 1981] J. N. Gray, P. R. McJones, and M. Blasgen, "The Recovery Manager of the System R Database Manager," *ACM Computing Surveys*, Volume 13, Number 2 (June 1981), pages 223–242.

[Grosshans 1986] D. Grosshans, *File Systems Design and Implementation*, Prentice-Hall, Englewood Cliffs, NJ (1986).

[Gupta and Franklin 1978] R. K. Gupta and M. A. Franklin, "Working Set and Page Fault Frequency Replacement Algorithms: A Performance Comparison," *IEEE Transactions on Computers*, Volume C-27, Number 8 (August 1978), pages 706–712.

[Habermann 1969] A. N. Habermann, "Prevention of System Deadlocks," *Communications of the ACM*, Volume 12, Number 7 (July 1969), pages 373–377 and 385.

[Hagmann 1989] R. Hagmann, "Comments on Workstation Operating Systems and Virtual Memory," *Proceedings of the Second Workshop on Workstation Operating Systems* (September 1989).

[Haldar and Subramanian 1991] S. Haldar and D. Subramanian, "Fairness in Processor Scheduling in Time Sharing Systems," *Operating Systems Review* (January 1991).

[Hall et al. 1980] D. E. Hall, D. K. Scherrer, and J. S. Sventek, "A Virtual Operating System," *Communications of the ACM*, Volume 23, Number 9 (September 1980), pages 495–502.

[Halsall 1992] F. Halsall, *Data Communications, Computer Networks, and Open Systems*, Addison-Wesley, Reading, MA (1992).

[Harker et al. 1981] J. M. Harker, D. W. Brede, R. E. Pattison, G. R. Santana, and L. G. Taft, "A Quarter Century of Disk File Innovation," *IBM Journal of Research and Development*, Volume 25, Number 5 (September 1981), pages 677–689.

[Harrison et al. 1976] M. A. Harrison, W. L. Ruzzo, and J. D. Ullman, "Protection in Operating Systems," *Communications of the ACM*, Volume 19, Number 8 (August 1976), pages 461–471.

[Havender 1968] J. W. Havender, "Avoiding Deadlock in Multitasking Systems," *IBM Systems Journal*, Volume 7, Number 2 (1968), pages 74–84.

[Hecht et al. 1988] M. S. Hecht, A. Johri, R. Aditham, and T. J. Wei, "Experience Adding C2 Security Features to UNIX," *Proceedings of the Summer 1988 USENIX Conference* (June 1988), pages 133–146.

[Hendricks and Hartmann 1979] E. C. Hendricks and T. C. Hartmann, "Evolution of a Virtual Machine Subsystem," *IBM Systems Journal*, Volume 18, Number 1 (1979), pages 111–142.

[Hennessy and Patterson 1990] J. L. Hennessy and D. A. Patterson, *Computer Architecture: a Quantitative Approach,* Morgan Kaufmann Publishers, Palo Alto, CA (1990).

[Henry 1984] G. Henry, "The Fair Share Scheduler," *AT&T Bell Laboratories Technical Journal* (October 1984).

[Hoagland 1985] A. S. Hoagland, "Information Storage Technology — A Look at the Future," *Computer,* Volume 18, Number 7 (July 1985), pages 60–68.

[Hoare 1972] C. A. R. Hoare, "Towards a Theory of Parallel Programming," in [Hoare and Perrott 1972], pages 61–71.

[Hoare 1974] C. A. R. Hoare, "Monitors: An Operating System Structuring Concept," *Communications of the ACM,* Volume 17, Number 10 (October 1974), pages 549–557; Erratum in *Communications of the ACM,* Volume 18, Number 2 (February 1975), page 95.

[Hoare and Perrott 1972] C. A. R. Hoare and R. H. Perrott, Eds., *Operating Systems Techniques,* Academic Press, London (1972).

[Hofri 1980] M. Hofri, "Disk Scheduling: FCFS Versus SSTF Revisited," *Communications of the ACM,* Volume 23, Number 11 (November 1980), pages 645–653.

[Holley et al. 1979] L. H. Holley, R. P. Parmelee, C. A. Salisbury, and D. N. Saul, "VM/370 Asymmetric Multiprocessing," *IBM Systems Journal,* Volume 18, Number 1 (1979), pages 47–70.

[Holt 1971] R. C. Holt, "Comments on Prevention of System Deadlocks," *Communications of the ACM,* Volume 14, Number 1 (January 1971), pages 36–38.

[Holt 1972] R. C. Holt, "Some Deadlock Properties of Computer Systems," *Computing Surveys,* Volume 4, Number 3 (September 1972), pages 179–196.

[Holt 1983] R. C. Holt, *Concurrent Euclid, the Unix System, and Tunis,* Addison-Wesley, Reading, MA (1983).

[Hong et al. 1989] J. Hong, X. Tan, and D. Towsley, "A Performance Analysis of Minimum Laxity and Earliest Deadline Scheduling in a Real-Time System," *IEEE Transactions on Computers,* Volume 38, Number 12 (December 1989) pages 1736–1744.

[Howard 1973] J. H. Howard, "Mixed Solutions for the Deadlock Problem," *Communications of the ACM,* Volume 16, Number 7 (July 1973), pages 427–430.

[Howard et al. 1988] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, and R. N. Sidebotham, "Scale and Performance in a Distributed File System," *ACM Transactions on Computer Systems,* Volume 6, Number 1 (February 1988), pages 55–81.

[Howarth et al. 1961] D. J. Howarth, R. B. Payne, and F. H. Sumner, "The Manchester University Atlas Operating System, Part II: User's Description," *Computer Journal,* Volume 4, Number 3 (October 1961), pages 226–229.

[Hsiao et al. 1979] D. K. Hsiao, D. S. Kerr, and S. E. Madnick, *Computer Security,* Academic Press, New York, NY (1979).

[Hyman 1985] D. Hyman, *The Columbus Chicken Statute, and More Bonehead Legislation,* S. Greene Press, Lexington, MA (1985).

[Iacobucci 1988] E. Iacobucci, *OS/2 Programmer's Guide,* Osborne McGraw-Hill, Berkeley, CA (1988).

[Iliffe and Jodeit 1962] J. K. Iliffe and J. G. Jodeit, "A Dynamic Storage Allocation System," *Computer Journal,* Volume 5, Number 3 (October 1962), pages 200–209.

[Intel 1985a] Intel Corporation, *iAPX 86/88, 186/188 User's Manual Programmer's Reference,* Intel Corp., Santa Clara, CA (1985).

[Intel 1985b] Intel Corporation, *iAPX 286 Programmer's Reference Manual,* Intel Corp., Santa Clara, CA (1985).

[Intel 1986] Intel Corporation, *iAPX 386 Programmer's Reference Manual,* Intel Corp., Santa Clara, CA (1986).

[Intel 1989] Intel Corporation, *i486 Microprocessor,* Intel Corp., Santa Clara, CA (1989).

[Intel 1990] Intel Corporation, *i486 Microprocessor Programmer's Reference Manual,* Intel Corp., Santa Clara, CA (1990).

[Isloor and Marsland 1980] S. S. Isloor and T. A. Marsland, "The Deadlock Problem: An Overview," *Computer,* Volume 13, Number 9 (September 1980), pages 58–78.

[ISO 1981] "ISO Open Systems Interconnection—Basic Reference Model," ISO/TC 97/SC 16 N 719, *International Organization for Standardization* (August 1981).

[Jensen et al. 1985] E. D. Jensen, C. D. Locke, and H. Tokuda, "A Time-Driven Scheduling Model for Real-Time Operating Systems," *Proceedings of the IEEE Real-Time Systems Symposium* (December 1985), pages 112–122.

[Jones 1978] A. K. Jones, "Protection Mechanisms and the Enforcement of Security Policies," in [Bayer et al. 1978], pages 228–250.

[Jones 1991] M. Jones, "Bringing the C Libraries With Us Into A Multi-Threaded Future," *Proceedings of the Winter 1991 USENIX Conference* (January 1991), pages 81–92.

[Jones and Liskov 1978] A. K. Jones and B. H. Liskov, "A Language Extension for Expressing Constraints on Data Access," *Communications of the ACM,* Volume 21, Number 5 (May 1978), pages 358–367.

[Jones and Schwarz 1980] A. K. Jones and P. Schwarz, "Experience Using Multiprocessor Systems — A Status Report," *Computing Surveys,* Volume 12, Number 2 (June 1980), pages 121–165.

[Jul et al. 1988] E. Jul, H. Levy, N. Hutchinson, and A. Black, "Fine-Grained Mobility in the Emerald System," *ACM Transactions on Computer Systems,* Volume 6, Number 1 (February 1988), pages 109–133.

[Katz et al. 1989] R.H. Katz, G.A. Gibson, and D.A. Patterson, "Disk System Architectures for High Performance Computing," *Proceedings of the IEEE,* Volume 77, Number 12 (December 1989).

[Kay and Lauder 1988] J. Kay and P. Lauder, "A Fair Share Scheduler," *Communications of the ACM,* Volume 31, Number 1 (January 1988), pages 44–55.

[Kenah el al. 1988] L. J. Kenah, R. E. Goldenberg, and S. F. Bate, *VAX/VMS Internals and Data Structures,* Digital Press, Bedford, MA (1988).

[Kenville 1982] R. F. Kenville, "Optical Disk Data Storage," *Computer,* Volume 15, Number 7 (July 1982), pages 21–26.

[Kernighan and Pike 1984] B. W. Kernighan and R. Pike, *The UNIX Programming Environment,* Prentice-Hall, Englewood Cliffs, NJ (1984).

[Kernighan and Ritchie 1988] B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Second Edition, Prentice-Hall, Englewood Cliffs, NJ (1988).

[Kessels 1977] J. L. W. Kessels, "An Alternative to Event Queues for Synchronization in Monitors," *Communications of the ACM,* Volume 20, Number 7 (July 1977), pages 500–503.

[Khanna et al. 1992] S. Khanna, M. Sebree, and J. Zolnowsky, "Realtime Scheduling in SunOS 5.0," *Proceedings of the Winter 1992 USENIX Conference,* San Francisco, CA (January 1992), pages 375–390.

[Kieburtz and Silberschatz 1978] R. B. Kieburtz and A. Silberschatz, "Capability Managers," *IEEE Transactions on Software Engineering,* Volume SE-4, Number 6 (November 1978), pages 467–477.

[Kieburtz and Silberschatz 1983] R. B. Kieburtz and A. Silberschatz, "Access Right Expressions," *ACM Transactions on Programming Languages and Systems,* Volume 5, Number 1 (January 1983), pages 78–96.

[Kilburn et al. 1961] T. Kilburn, D. J. Howarth, R. B. Payne, and F. H. Sumner, "The Manchester University Atlas Operating System, Part I: Internal Organization," *Computer Journal,* Volume 4, Number 3 (October 1961), pages 222–225.

[Kleinrock 1975] L. Kleinrock, *Queueing Systems, Volume II: Computer Applications,* Wiley-Interscience, New York, NY (1975).

[Knapp 1987] E. Knapp, "Deadlock Detection in Distributed Databases," *Computing Surveys*, Volume 19, Number 4 (December 1987), pages 303–328.

[Knuth 1966] D. E. Knuth, "Additional Comments on a Problem in Concurrent Programming Control," *Communications of the ACM*, Volume 9, Number 5 (May 1966), pages 321–322.

[Knuth 1973] D. E. Knuth, *The Art of Computer Programming, Volume 1: Fundamental Algorithms*, Second Edition, Addison-Wesley, Reading, MA (1973).

[Koch 1987] P. D. L. Koch, "Disk File Allocation Based on the Buddy System," *ACM Transactions on Computer Systems*, Volume 5, Number 4 (November 1987), pages 352–370.

[Kogan and Rawson 1988] M. S. Kogan and F. L. Rawson, "The Design of Operating System/2," *IBM Systems Journal*, Volume 27, Number 2 (1988), pages 90–104.

[Korn 1983] D. Korn, "KSH, A Shell Programming Language," *Proceedings of the Summer 1983 USENIX Conference* (July 1983), pages 191–202.

[Korth and Silberschatz 1991] H. F. Korth and A. Silberschatz, *Database System Concepts*, Second Edition, McGraw Hill, New York, NY (1991).

[Kosaraju 1973] S. Kosaraju, "Limitations of Dijkstra's Semaphore Primitives and Petri Nets," *Operating Systems Review*, Volume 7, Number 4 (October 1973), pages 122–126.

[Krakowiak 1988] S. Krakowiak, *Principles of Operating Systems*, MIT Press, Cambridge, MA (1988).

[Kramer 1988] S. M. Kramer, "Retaining SUID Programs in a Secure UNIX," *Proceedings of the 1988 Summer USENIX Conference* (June 1988), pages 107–118.

[Lamport 1974] L. Lamport, "A New Solution of Dijkstra's Concurrent Programming Problem," *Communications of the ACM*, Volume 17, Number 8 (August 1974), pages 453–455.

[Lamport 1976] L. Lamport, "Synchronization of Independent Processes," *Acta Informatica*, Volume 7, Number 1 (1976), pages 15–34.

[Lamport 1977] L. Lamport, "Concurrent Reading and Writing," *Communications of the ACM*, Volume 20, Number 11 (November 1977), pages 806–811.

[Lamport 1978a] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, Volume 21, Number 7 (July 1978), pages 558–565.

[Lamport 1978b] L. Lamport, "The Implementation of Reliable Distributed Multiprocess Systems," *Computer Networks*, Volume 2, Number 2 (April 1978), pages 95–114.

[Lamport 1981] L. Lamport, "Password Authentication with Insecure Communications," *Communications of the ACM,* Volume 24, Number 11 (November 1981), pages 770–772.

[Lamport 1986] L. Lamport, "The Mutual Exclusion Problem," *Journal of the ACM,* Volume 33, Number 2 (1986), pages 313–348.

[Lamport 1991] L. Lamport, "The Mutual Exclusion Problem Has Been Solved," *Communications of the ACM,* Volume 34, Number 1 (January 1991), page 110.

[Lamport et al. 1982] L. Lamport, R. Shostak, and M. Pease, "The Byzantine Generals Problem," *ACM Transactions on Programming Languages and Systems,* Volume 4, Number 3 (July 1982), pages 382–401.

[Lampson 1968] B. W. Lampson, "A Scheduling Philosophy for Multiprocessing Systems," *Communications of the ACM,* Volume 11, Number 5 (May 1968), pages 347–360.

[Lampson 1969] B. W. Lampson, "Dynamic Protection Structures," *Proceedings of the AFIPS Fall Joint Computer Conference* (1969), pages 27–38.

[Lampson 1971] B. W. Lampson, "Protection," *Proceedings of the Fifth Annual Princeton Conference on Information Science Systems* (1971), pages 437–443; reprinted in *Operating System Review,* Volume 8, Number 1 (January 1974), pages 18–24.

[Lampson 1973] B. W. Lampson, "A Note on the Confinement Problem," *Communications of the ACM,* Volume 10, Number 16 (October 1973), pages 613–615.

[Lampson and Sturgis 1976] B. Lampson and H. Sturgis, "Crash Recovery in a Distributed Data Storage System," Technical Report, Computer Science Laboratory, Xerox, Palo Alto Research Center, Palo Alto, CA (1976).

[Landwehr 1981] C. E. Landwehr, "Formal Models of Computer Security," *Computing Surveys,* Volume 13, Number 3 (September 1981), pages 247–278.

[Larson and Kajla 1984] P. Larson and A. Kajla, "File Organization: Implementation of a Method Guaranteeing Retrieval in One Access," *Communications of the ACM,* Volume 27, Number 7 (July 1984), pages 670–677.

[Lazowska et al. 1984] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik, *Quantitative System Performance,* Prentice-Hall, Englewood Cliffs, NJ (1984).

[Leach et al. 1982] P. J. Leach, B. L. Stump, J. A. Hamilton, and P. H. Levine, "UID's as Internal Names in a Distributed File System," *Proceedings of the First Symposium on Principles of Distributed Computing* (August 1982), pages 34–41.

[Leffler et al. 1978] S. J. Leffler, R. S. Fabry, and W. N. Joy, "A 4.2BSD Interprocess Communication Primer," *Unix Programmer's Manual,* Volume 2C, University of California at Berkeley, CA (1978).

[Leffler et al. 1983] S. J. Leffler, W. N. Joy, and R. S. Fabry, "4.2BSD Networking Implementation Notes," *Unix Programmer's Manual,* Volume 2C, University of California at Berkeley, CA (1983).

[Leffler et al. 1989] S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System,* Addison-Wesley, Reading, MA (1989).

[Lehmann 1987] F. Lehmann, "Computer Break-Ins," *Communications of the ACM,* Volume 30, Number 7 (July 1987), pages 584–585.

[Le Lann 1977] G. Le Lann, "Distributed Systems — Toward a Formal Approach," *Proceedings of the IFIP Congress 77* (1977), pages 155–160.

[Lempel 1979] A. Lempel, "Cryptology in Transition," *Computing Surveys,* Volume 11, Number 4 (December 1979), pages 286–303.

[Lett and Konigsford 1968] A. L. Lett and W. L. Konigsford, "TSS/360: A Time-Shared Operating System," *Proceedings of the AFIPS Fall Joint Computer Conference* (1968), pages 15–28.

[Leutenegger and Vernon 1990] S. Leutenegger and M. Vernon, "The Performance of Multiprogrammed Multiprocessor Scheduling Policies," *Proceedings of the Conference on Measurement and Modeling of Computer Systems* (May 1990).

[Letwin 1988] G. Letwin, *Inside OS/2,* Microsoft Press, Redmond, WA (1988).

[Levin et al. 1975] R. Levin, E. S. Cohen, W. M. Corwin, F. J. Pollack, and W. A. Wulf, "Policy/Mechanism Separation in Hydra," *Proceedings of the Fifth ACM Symposium on Operating System Principles* (1975), pages 132–140.

[Levy and Lipman 1982] H. M. Levy and P. H. Lipman, "Virtual Memory Management in the VAX/VMS Operating System," *Computer,* Volume 15, Number 3 (March 1982), pages 35–41.

[Levy and Silberschatz 1990] E. Levy and A. Silberschatz, "Distributed File Systems: Concepts and Examples," *Computing Surveys,* Volume 22, Number 4 (December 1990), pages 321–374.

[Lichtenberger and Pirtle 1965] W. W. Lichtenberger and M. W. Pirtle, "A Facility for Experimentation in Man-Machine Interaction," *Proceedings of the AFIPS Fall Joint Computer Conference* (1965), pages 589–598.

[Lions 1977] J. Lions, "A Commentary on the UNIX Operating System," Technical Report, Carnegie-Mellon University (April 1988).

[Lipner 1975] S. Lipner, "A Comment on the Confinement Problem," *Operating System Review,* Volume 9, Number 5 (November 1975), pages 192–196.

[Lipton 1974] R. Lipton, "On Synchronization Primitive Systems," PhD. Thesis, Carnegie-Mellon University (1974).

[Liskov 1972] B. H. Liskov, "The Design of the Venus Operating System," *Communications of the ACM,* Volume 15, Number 3 (March 1972), pages 144–149.

[Litzkow et al. 1988] M. J. Litzkow, M. Livny, and M. W. Mutka, "Condor—A Hunter of Idle Workstations," *Proceedings of the Eighth IEEE International Conference on Distributed Computing Systems* (1988), pages 104–111.

[Liu and Layland 1973] C. L. Liu and J. W. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," *Journal of the ACM,* Volume 20, Number 1 (January 1973), pages 46–61.

[Lobel 1986] J. Lobel, *Foiling the System Breakers: Computer Security and Access Control,* McGraw-Hill, New York, NY (1986).

[Loepere 1992] K. Loepere, "Mach 3 Kernel Principles," Technical Report, Open Software Foundation, MA (January 1992).

[Loucks and Sauer 1987] L. K. Loucks and C. H. Sauer, "Advanced Interactive Executive (AIX) Operating System Overview," *IBM Systems Journal,* Volume 26, Number 4 (1987), pages 326–345.

[Lynch 1972] W. C. Lynch, "An Operating System Design for the Computer Utility Environment," in [Hoare and Perrott 1972], pages 341–350.

[MacKinnon 1979] R. A. MacKinnon, "The Changing Virtual Machine Environment: Interfaces to Real Hardware, Virtual Hardware, and Other Virtual Machines," *IBM Systems Journal,* Volume 18, Number 1 (1979), pages 18–46.

[Maekawa 1985] M. Maekawa, "A Square Root Algorithm for Mutual Exclusion in Decentralized Systems," *ACM Transactions on Computer Systems,* Volume 3, Number 2 (May 1985), pages 145–159.

[Maekawa et al. 1987] M. Maekawa, A. E. Oldehoeft, and R. R. Oldehoeft, *Operating Systems: Advanced Concepts,* Benjamin/Cummings, Menlo Park, CA (1987).

[Maples 1985] C. Maples, "Analyzing Software Performance in a Multiprocessor Environment," *IEEE Software,* Volume 2, Number 4 (July 1985).

[Marsh et al. 1991] B. D. Marsh, M. L. Scott, T. J. LeBlanc, and E. P. Markatos, "First-Class User-Level Threads," *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principals* (October 1991), pages 110–121.

[Massalin and Pu 1989] H. Massalin and C. Pu, "Threads and Input/Output in the Synthesis Kernel," *Proceedings of the 12th Symposium on Operating Systems Principles* (December 1989), pages 191–200.

[Mattson et al. 1970] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, "Evaluation Techniques for Storage Hierarchies," *IBM Systems Journal,* Volume 9, Number 2 (1970), pages 78–117.

[McGraw and Andrews 1979] J. R. McGraw and G. R. Andrews, "Access Control in Parallel Programs," *IEEE Transactions on Software Engineering,* Volume SE-5, Number 1 (January 1979), pages 1–9.

[McKeag and Wilson 1976] R. M. McKeag and R. Wilson, *Studies in Operating Systems,* Academic Press, London (1976).

[McKeon 1985] B. McKeon, "An Algorithm for Disk Caching with Limited Memory," *Byte,* Volume 10, Number 9 (September 1985), pages 129–138.

[McKusick and Karels 1988] M. K. McKusick and K. J. Karels, "Design of a General Purpose Memory Allocator for the 4.3BSD UNIX Kernel," *Proceedings of the Summer 1988 USENIX Conference* (June 1988), pages 295–304.

[McKusick et al. 1984] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. "A Fast File System for UNIX," *ACM Transactions on Computer Systems,* Volume 2, Number 3 (August 1984), pages 181–197.

[McNamee and Armstrong 1990] D. McNamee and K. Armstrong, "Extending the Mach External Pager Interface to Accommodate User-Level Page-Replacement Policies," *Proceedings of the USENIX MACH Workshop,* Burlington, VT (October 1990).

[McVoy and Kleiman 1991] L. W. McVoy and S. R. Kleiman, "Extent-like Performance from a UNIX File System," *Proceedings of the Winter 1991 USENIX Conference* (January 1991), pages 33–44.

[Mealy et al. 1966] G. H. Mealy, B. I. Witt, and W. A. Clark, "The Functional Structure of OS/360," *IBM Systems Journal,* Volume 5, Number 1 (1966).

[Menasce and Muntz 1979] D. Menasce and R. R. Muntz, "Locking and Deadlock Detection in Distributed Data Bases," *IEEE Transactions on Software Engineering,* Volume SE-5, Number 3 (May 1979), pages 195–202.

[Metzner 1982] J. R. Metzner, "Structuring Operating Systems Literature for the Graduate Course," *Operating Systems Review,* Volume 16, Number 4 (October 1982), pages 10–25.

[Meyer and Seawright 1970] R. A. Meyer and L. H. Seawright, "A Virtual Machine Time-Sharing System," *IBM Systems Journal,* Volume 9, Number 3 (1970), pages 199–218.

[Microsoft 1986] Microsoft Corporation, *Microsoft MS-DOS User's Reference* and *Microsoft MS-DOS Programmer's Reference,* Microsoft Press, Redmond, WA (1986).

[Microsoft 1989] Microsoft Corporation, *Microsoft Operating System/2 Programmer's Reference,* 3 Volumes, Microsoft Press, Redmond, WA (1989).

[Microsoft 1991] Microsoft Corporation, *Microsoft MS-DOS User's Guide and Reference,* Microsoft Press, Redmond, WA (1991).

[Milenkovic 1987] M. Milenkovic, *Operating Systems: Concepts and Design,* McGraw-Hill, New York, NY (1987).

[Mohan and Lindsay 1983] C. Mohan and B. Lindsay, "Efficient Commit Protocols for the Tree of Processes Model of Distributed Transactions," *Proceedings of the Second ACM SIGACT-SIGOPS Symposium on the Principles of Distributed Computing* (1983).

[Morris 1973] J. H. Morris, "Protection in Programming Languages," *Communications of the ACM,* Volume 16, Number 1 (January 1973), pages 15–21.

[Morris et al. 1986] J. H. Morris, M. Satyanarayanan, M. H. Conner, J. H. Howard, D. S. H. Rosenthal, and F. D. Smith, "Andrew: A Distributed Personal Computing Environment," *Communications of the ACM,* Volume 29, Number 3 (March 1986), pages 184–201.

[Morris and Thompson 1979] R. Morris and K. Thompson, "Password Security: A Case History," *Communications of the ACM,* Volume 22, Number 11 (November 1979), pages 594–597.

[Morshedian 1986] D. Morshedian, "How to Fight Password Pirates," *Computer,* Volume 19, Number 1 (January 1986).

[Motorola 1989a] Motorola Inc., *MC68000 Family Reference,* Second Edition, Prentice-Hall, Englewood Cliffs, NJ (1989).

[Motorola 1989b] Motorola Inc., *MC68030 Enhanced 32-Bit Microprocessor User's Manual,* Second Edition, Prentice-Hall, Englewood Cliffs, NJ (1989).

[Mullender and Tanenbaum 1985] S. J. Mullender and A. S. Tanenbaum, "A Distributed File Service Based on Optimistic Concurrency Control," *Proceedings of the Tenth Symposium on Operating Systems Principles* (December 1985), pages 51–62.

[Mullender et al. 1990] S. J. Mullender, G. Van Rossum, A. S. Tanenbaum, R. Van Renesse, and H. Van Staveren, "Amoeba: A Distributed-Operating System for the 1990s," *IEEE Computer,* Volume 23, Number 5 (May 1990), pages 44–53.

[Mutka and Livny 1987] M. W. Mutka and M. Livny, "Scheduling Remote Processor Capacity in a Workstation-Processor Bank Network," *Proceedings of the Seventh IEEE International Conference on Distributed Computing Systems* (1987).

[Needham and Walker 1977] R. M. Needham and R. D. H. Walker, "The Cambridge CAP Computer and its Protection System," *Proceedings of the Sixth Symposium on Operating System Principles* (November 1977), pages 1–10.

[Nelson and Cheng 1992] B. Nelson and Y. Cheng, "How and Why SCSI is better than IPI for NFS," *Proceedings of the Winter 1992 USENIX Conference,* San Francisco, CA, (January 1992), pages 253–270.

[Nelson et al. 1988] M. Nelson, B. Welch, and J. K. Ousterhout, "Caching in the Sprite Network File System," *ACM Transactions on Computer Systems,* Volume 6, Number 1 (February 1988), pages 134–154.

[Newton 1979] G. Newton, "Deadlock Prevention, Detection, and Resolution: An Annotated Bibliography," *Operating Systems Review,* Volume 13, Number 2 (April 1979), pages 33–44.

[Nichols 1987] D. A. Nichols, "Using Idle Workstations in a Shared Computing Environment," *Proceedings of the Eleventh ACM Symposium on Operating Systems Principals* (October 1987), pages 5–12.

[Norton 1986] P. Norton, *Inside the IBM PC,* Revised and Enlarged, Brady Books, New York, NY (1986).

[Norton and Wilton 1988] P. Norton and R. Wilton, *The New Peter Norton Programmer's Guide to the IBM PC & PS/2,* Microsoft Press, Redmond, WA (1988).

[O'Leary and Kitts 1985] B. T. O'Leary and D. L. Kitts, "Optical Device for a Mass Storage System," *Computer,* Volume 18, Number 7 (July 1985).

[Obermarck 1982] R. Obermarck, "Distributed Deadlock Detection Algorithm," *ACM Transactions on Database Systems,* Volume 7, Number 2 (June 1982), pages 187–208.

[Oldehoeft and Allan 1985] R. R. Oldehoeft and S. J. Allan, "Adaptive Exact-Fit Storage Management," *Communications of the ACM,* Volume 28, Number 5 (May 1985), pages 506–511.

[Olsen and Kenley 1989] R. P. Olsen and G. Kenley, "Virtual Optical Disks Solve the On-Line Storage Crunch," *Computer Design,* Volume 28, Number 1 (January 1989), pages 93–96.

[Organick 1972] E. I. Organick, *The Multics System: An Examination of its Structure,* MIT Press, Cambridge, MA (1972).

[OSF 1989] Open Software Foundation, *Mach Technology: A series of Ten Lectures,* OSF Foundation, Cambridge MA (Fall 1989).

[Ousterhout et al. 1988] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch, "The Sprite Network-Operating System," *IEEE Computer,* Volume 21, Number 2 (February 1988), pages 23–36.

[Parmelee et al. 1972] R. P. Parmelee, T. I. Peterson, C. C. Tillman, and D. Hatfield, "Virtual Storage and Virtual Machine Concepts," *IBM Systems Journal,* Volume 11, Number 2 (1972), pages 99–130.

[Parnas 1975] D. L. Parnas, "On a Solution to the Cigarette Smokers' Problem Without Conditional Statements," *Communications of the ACM,* Volume 18, Number 3 (March 1975), pages 181–183.

[Parnas and Habermann 1972] D. L. Parnas and A. N. Habermann, "Comment on Deadlock Prevention Method," *Communications of the ACM,* Volume 15, Number 9 (September 1972), pages 840–841.

[Patil 1971] S. Patil, "Limitations and Capabilities of Dijkstra's Semaphore Primitives for Coordination Among Processes," Technical Report, MIT (1971).

[Patterson et al. 1988] D. A. Patterson, G. Gibson, and R. H. Katz, "A Case for Redundant Arrays of Inexpensive Disks (RAID)," *Proceedings of the ACM SIGMOD International Conference on the Management of Data* (1988).

[Peacock 1992] J. K. Peacock, "File System Multithreading in System V Release 4 MP," *Proceedings of the Summer 1992 USENIX Conference* (June 1992), pages 19–29.

[Pease et al. 1980] M. Pease, R. Shostak, and L. Lamport, "Reaching Agreement in the Presence of Faults," *Journal of the ACM,* Volume 27, Number 2 (April 1980), pages 228–234.

[Pechura and Schoeffler 1983] M. A. Pechura and J. D. Schoeffler, "Estimating File Access Time of Floppy Disks," *Communications of the ACM,* Volume 26, Number 10 (October 1983), pages 754–763.

[Peterson 1981] G. L. Peterson, "Myths About the Mutual Exclusion Problem," *Information Processing Letters,* Volume 12, Number 3 (June 1981).

[Pfleeger 1989] C. Pfleeger, *Security in Computing,* Prentice-Hall, Englewood Cliffs, NJ (1989).

[Pinkert and Wear 1989] J. Pinkert and L. Wear, *Operating Systems: Concepts, Policies, and Mechanisms,* Prentice-Hall, Englewood Cliffs, NJ (1989).

[Popek 1974] G. J. Popek, "Protection Structures," *Computer,* Volume 7, Number 6 (June 1974), pages 22–33.

[Popek and Walker 1985] G. Popek and B. Walker, Eds., *The LOCUS Distributed System Architecture,* MIT Press, Cambridge, MA (1985).

[Powell et al. 1991] M. L. Powell, S. R. Kleiman, S. Barton, D. Shaw, D. Stein, and M. Weeks, "SunOS Multi-threaded Architecture," *Proceedings of the Winter 1991 USENIX Conference* (January 1991), pages 65–80.

[Prieve and Fabry 1976] B. G. Prieve and R. S. Fabry, "VMIN — An Optimal Variable Space Page-Replacement Algorithm," *Communications of the ACM,* Volume 19, Number 5 (May 1976), pages 295–297.

[Purdin et al. 1987] T. D. M. Purdin, R. D. Schlichting, and G. R. Andrews, "A File Replication Facility for Berkeley UNIX," *Software—Practice and Experience,* Volume 17 (December 1987), pages 923–940.

[Quarterman 1990] J. S. Quarterman, *The Matrix Computer Networks and Conferencing Systems Worldwide,* Digital Press, Bedford, MA (1990).

[Quarterman et al. 1985] J. S. Quarterman, A. Silberschatz, and J. L. Peterson, "4.2BSD and 4.3BSD as Examples of the UNIX Systems," *Computing Surveys,* Volume 17, Number 4 (December 1985), pages 379–418.

[Quarterman and Hoskins 1986] J. S. Quarterman and H. C. Hoskins, "Notable Computer Networks," *Communications of the ACM,* Volume 29, Number 10 (October 1986), pages 932–971.

[Rashid 1986] R. F. Rashid, "From RIG to Accent to Mach: The Evolution of a Network-Operating System," *Proceedings of the ACM/IEEE Computer Society, Fall Joint Computer Conference* (1986).

[Rashid and Robertson 1981] R. Rashid and G. Robertson, "Accent: A Communication Oriented Network-Operating System Kernel," *Proceedings of the Eighth Symposium on Operating System Principles* (December 1981).

[Raynal 1986] M. Raynal, *Algorithms for Mutual Exclusion,* MIT Press, Cambridge, MA (1986).

[Raynal 1991] M. Raynal, "A Simple Taxonomy for Distributed Mutual Exclusion Algorithms," *Operating Systems Review,* Volume 25 (April 1991), pages 47–50.

[Redell and Fabry 1974] D. D. Redell and R. S. Fabry, "Selective Revocation of Capabilities," *Proceedings of the IRIA International Workshop on Protection in Operating Systems* (1974), pages 197–210.

[Reed 1983] D. P. Reed, "Implementing Atomic Actions on Decentralized Data," *ACM Transactions on Computer Systems,* Volume 1 (February 1983), pages 3–23.

[Reed and Kanodia 1979] D. P. Reed and R. K. Kanodia, "Synchronization with Eventcounts and Sequences," *Communications of the ACM,* Volume 22, Number 2 (February 1979), pages 115–123.

[Reid 1987] B. Reid, "Reflections on Some Recent Widespread Computer Break-Ins," *Communications of the ACM,* Volume 30, Number 2 (February 1987), pages 103–105.

[Ricart and Agrawala 1981] G. Ricart and A. K. Agrawala, "An Optimal Algorithm for Mutual Exclusion in Computer Networks," *Communications of the ACM,* Volume 24, Number 1 (January 1981), pages 9–17.

[Ritchie 1979] D. Ritchie, "The Evolution of the UNIX Time-Sharing System," *Language Design and Programming Methodology, Lecture Notes on Computer Science,* Volume 79, Springer-Verlag, Berlin (1979).

[Ritchie and Thompson 1974] D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," *Communications of the ACM,* Volume 17, Number 7 (July 1974), pages 365–375; a later version appeared in *Bell System Technical Journal,* Volume 57, Number 6 (July-August 1978), pages 1905–1929.

[Rivest et al. 1978] R. L. Rivest, A. Shamir, and L. Adleman, "On Digital Signatures and Public Key Cryptosystems," *Communications of the ACM,* Volume 21, Number 2 (February 1978), pages 120–126.

[Rosen 1969] S. Rosen, "Electronic Computers: A Historical Survey," *Computing Surveys,* Volume 1, Number 1 (March 1969), pages 7–36.

[Rosenblum and Ousterhout 1991] M. Rosenblum and J. K. Ousterhout, "The Design and Implementation of a Log-Structured File System," *Proceedings of the 13th ACM Symposium on Operating Systems Principles* (October 1991), pages 1–15.

[Rosenkrantz et al. 1978] D. J. Rosenkrantz, R. E. Stearns, and P. M. Lewis, II, "System Level Concurrency Control for Distributed Database Systems," *ACM Transactions on Database Systems,* Volume 3, Number 2 (June 1978), pages 178–198.

[Ruschitzka and Fabry 1977] M. Ruschizka and R. S. Fabry, "A Unifying Approach to Scheduling," *Communications of the ACM,* Volume 20, Number 7 (July 1977), pages 469–477.

[Rushby 1981] J. M. Rushby, "Design and Verification of Secure Systems," *Proceedings of the Eighth Symposium on Operating System Principles* (December 1981), pages 12–21.

[Rushby and Randell 1983] J. Rushby and B. Randell, "A Distributed Secure System," *Computer,* Volume 16, Number 7 (July 1983), pages 55–67.

[Russell and Gangemi 1991] D. Russell and G. T. Gangemi, Sr., *Computer Security Basics,* O'Reilly & Associates, Inc., Sebastopol, CA (1991).

[Saltzer and Schroeder 1975] J. H. Saltzer and M. D. Schroeder, "The Protection of Information in Computer Systems," *Proceedings of the IEEE,* Volume 63, Number 9 (September 1975), pages 1278–1308.

[Samson 1990] S. Samson, *MVS Performance Management,* McGraw-Hill, New York, NY (1990).

[Sandberg 1987] R. Sandberg, *The Sun Network File System: Design, Implementation, and Experience,* Sun Microsystems, Inc., Mountain View, CA (1987).

[Sandberg et al. 1985] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon, "Design and Implementation of the Sun Network Filesystem," *Proceedings of the 1985 USENIX Summer Conference* (June 1985), pages 119–130.

[Sanguinetti 1986] J. Sanguinetti, "Performance of a Message-Based Multiprocessor," *Computer,* Volume 19, Number 9 (September 1986), pages 47–56.

[Sarisky 1983] L. Sarisky, "Will Removable Hard Disks Replace the Floppy?" *Byte* (March 1983), pages 110–117.

[Satyanarayanan 1980] M. Satyanarayanan, *Multiprocessors: A Comparative Study,* Prentice-Hall, Englewood Cliffs, NJ (1980).

[Satyanarayanan 1989] M. Satyanarayanan, "Integrating Security in a Large Distributed System," *ACM Transactions on Computer Systems,* Volume 7 (August 1989), pages 247–280.

[Satyanarayanan 1990] M. Satyanarayanan, "Scalable, Secure, and Highly Available Distributed File Access," *Computer,* Volume 23, Number 5 (May 1990), pages 9–21.

[Sauer and Chandy 1981] C. H. Sauer and K. M. Chandy, *Computer Systems Performance Modeling,* Prentice-Hall, Englewood Cliffs, NJ (1981).

[Schell 1983] R. R. Schell, "A Security Kernel for a Multiprocessor Microcomputer," *Computer,* Volume 16, Number 7 (July 1983), pages 47–53.

[Schlichting and Schneider 1982] R. D. Schlichting and F. B. Schneider, "Understanding and Using Asynchronous Message Passing Primitives," *Proceedings of the Symposium on Principles of Distributed Computing* (August 1982), pages 141–147.

[Schneider 1982] F. B. Schneider, "Synchronization in Distributed Programs," *ACM Transactions on Programming Languages and Systems,* Volume 4, Number 2 (April 1982), pages 125–148.

[Schrage 1967] L. E. Schrage, "The Queue M/G/I with Feedback to Lower Priority Queues," *Management Science,* Volume 13 (1967), pages 466–474.

[Schroeder et al. 1985] M. D. Schroeder, D. K. Gifford, and R. M. Needham, "A Caching File System for a Programmer's Workstation," *Proceedings of the Tenth Symposium on Operating Systems Principles* (December 1985), pages 25–32.

[Schultz 1988] B. Schultz, "VM: The Crossroads of Operating Systems," *Datamation,* Volume 34, Number 14 (July 1988), pages 79–84.

[Schwartz and Weissman 1967] J. I. Schwartz and C. Weissman, "The SDC Time-Sharing System Revisited," *Proceedings of the ACM National Meeting* (August 1967), pages 263–271.

[Schwartz et al. 1964] J. I. Schwartz, E. G. Coffman, and C. Weissman, "A General Purpose Time-Sharing System," *Proceedings of the AFIPS Spring Joint Computer Conference* (1964), pages 397–411.

[Seawright and MacKinnon 1979] L. H. Seawright and R. A. MacKinnon, "VM/370 — A Study of Multiplicity and Usefulness," *IBM Systems Journal,* Volume 18, Number 1 (1979), pages 4–17.

[Seely 1989] D. Seely, "Password Cracking: A Game of Wits," *Communications of the ACM,* Volume 32, Number 6 (June 1989), pages 700–704.

[Shore 1975] J. E. Shore, "On the External Storage Fragmentation Produced by First-Fit and Best-Fit Allocation Strategies," *Communications of the ACM,* Volume 18, Number 8 (August 1975), pages 433–440.

[Shrivastava and Panzieri 1982] S. K. Shrivastava and F. Panzieri, "The Design of a Reliable Remote Procedure Call Mechanism," *IEEE Transactions on Computers,* Volume C-31, Number 7 (July 1982), pages 692–697.

[Silverman 1983] J. M. Silverman, "Reflections on the Verification of the Security of an Operating System Kernel," *Proceedings of the Ninth Symposium on Operating Systems Principles,* Volume 17, Number 5 (October 1983), pages 143–154.

[Simmons 1979] G. J. Simmons, "Symmetric and Asymmetric Encryption," *Computing Surveys,* Volume 11, Number 4 (December 1979), pages 304–330.

[Singhal 1989] M. Singhal, "Deadlock Detection in Distributed Systems," *IEEE Computer,* Volume 22, Number 11 (November 1989), pages 37–48.

[Smith 1982] A. J. Smith, "Cache Memories," *ACM Computing Surveys,* Volume 14, Number 3 (September 1982), pages 473–530.

[Smith 1985] A. J. Smith, "Disk Cache-Miss Ratio Analysis and Design Considerations," *ACM Transactions on Computer Systems,* Volume 3, Number 3 (August 1985), pages 161–203.

[Smith 1988] A. J. Smith, "A Survey of Process Migration Mechanisms," *Operating Systems Review* (July 1988).

[Spafford 1989] E. H. Spafford, "The Internet Worm: Crisis and Aftermath," *Communications of the ACM,* Volume 32, Number 6 (June 1989), pages 678–687.

[Spector and Schwarz 1983] A. Z. Spector and P. M. Schwarz, "Transactions: A Construct for Reliable Distributed Computing," *ACM SIGOPS Operating Systems Review*, Volume 17, Number 2 (1983), pages 18–35.

[Stallings 1984] W. Stallings, "Local Networks," *Computing Surveys,* Volume 16, Number 1 (March 1984), pages 1–41.

[Stallings 1992] W. Stallings, *Operating Systems,* Macmillan, New York (1992).

[Stankovic 1982] J. S. Stankovic, "Software Communication Mechanisms: Procedure Calls Versus Messages," *Computer,* Volume 15, Number 4 (April 1982).

[Stankovic and Ramamrithan 1989] J. S. Stankovic and K. Ramamrithan, "The Spring Kernel: A New Paradigm for Real-Time Operating Systems," *Operating Systems Review* (July 1989).

[Staunstrup 1982] J. Staunstrup, "Message Passing Communication versus Procedure Call Communication," *Software—Practice and Experience,* Volume 12, Number 3 (March 1982), pages 223–234.

[Stein and Shaw 1992] D. Stein and D. Shaw, "Implementing Lightweight Threads," *Proceedings of the Summer 1992 USENIX Conference* (June 1992), pages 1–9.

[Stephenson 1983] C. J. Stephenson, "Fast Fits: A New Method for Dynamic Storage Allocation," *Proceedings of the Ninth Symposium on Operating Systems Principles* (December 1983), pages 30–32.

[Stevens 1990] W. R. Stevens, *UNIX Network Programming,* Prentice-Hall, Englewood Cliffs, NJ (1990).

[Stevens 1992] W. R. Stevens, *Advanced Programming in the UNIX Environment,* Addison-Wesley, Reading, MA (1992).

[Strachey 1959] C. Strachey, "Time Sharing in Large Fast Computers," *Proceedings of the International Conference on Information Processing* (June 1959), pages 336–341.

[Sun Microsystems 1990] Sun Microsystems, *Network Programming Guide,* Sun Microsystems, Inc., Mountain View, CA (1990), pages 168–186.

[Svobodova 1976] L. Svobodova, *Computer Performance Measurement and Evaluation,* Elsevier North-Holland, New York, NY (1976).

[Svobodova 1984] L. Svobodova, "File Servers for Network-Based Distributed Systems," *ACM Computing Surveys,* Volume 16, Number 4 (December 1984), pages 353–398.

[Tabak 1990] D. Tabak, *Multiprocessors,* Prentice-Hall, Englewood Cliffs, NJ (1990).

[Tanenbaum 1987] A. S. Tanenbaum, *Operating System Design and Implementation,* Prentice-Hall, Englewood Cliffs, NJ (1987).

[Tanenbaum 1988] A. S. Tanenbaum, *Computer Networks,* Second Edition, Prentice-Hall, Englewood Cliffs, NJ (1988).

[Tanenbaum 1990] A. S. Tanenbaum, *Structured Computer Organization,* Third Edition, Prentice-Hall, Englewood Cliffs, NJ (1990).

[Tanenbaum 1992] A. S. Tanenbaum, *Modern Operating Systems,* Prentice-Hall, Englewood Cliffs, NJ (1992).

[Tanenbaum and Van Renesse 1985] A. S. Tanenbaum and R. Van Renesse, "Distributed-Operating Systems," *ACM Computing Surveys,* Volume 17, Number 4 (December 1985), pages 419–470.

[Tanenbaum et al. 1990] A. S. Tanenbaum, R. Van Renesse, H. Van Staveren, G. J. Sharp, S. J. Mullender, J. Jansen, G. Van Rossum, "Experiences with the Amoeba Distributed-Operating System," *Communications of the ACM,* Volume 33, Number 12 (December 1990), pages 46–63.

[Tay and Ananda 1990] B. H. Tay and A. L. Ananda, "A Survey of Remote Procedure Calls," *Operating Systems Review,* Volume 24, Number 3 (July 1990), pages 68–79.

[Teorey and Pinkerton 1972] T. J. Teorey and T. B. Pinkerton, "A Comparative Analysis of Disk Scheduling Policies," *Communications of the ACM,* Volume 15, Number 3 (March 1972), pages 177–184.

[Tevanian et al. 1987a] A. Tevanian, Jr., R. F. Rashid, D. B. Golub, D. L. Black, E. Cooper, and M. W. Young, "Mach Threads and the Unix Kernel: The Battle for Control," *Proceedings of the Summer 1987 USENIX Conference* (July 1987).

[Tevanian et al. 1987b] A. Tevanian, Jr., R. F. Rashid, M. W. Young, D. B. Golub, M. R. Thompson, W. Bolosky, and R. Sanzi, "A UNIX Interface for Shared Memory and Memory Mapped Files Under Mach," Technical Report, Carnegie-Mellon University, Pittsburgh, PA (July 1987).

[Tevanian and Smith 1989] A. Tevanian and B. Smith, "Mach: The Model for Future Unix," *Byte* (November 1989).

[Thompson 1978] K. Thompson, "UNIX Implementation," *The Bell System Technical Journal,* Volume 57, Number 6, Part 2 (July-August 1978), pages 1931–1946.

[Thurber and Freeman 1980] K. J. Thurber and H. A. Freeman, "Updated Bibliography on Local Computer Networks," *Computer Architecture News,* Volume 8 (April 1980), pages 20–28.

[Traiger et al. 1982] I. L. Traiger, J. N. Gray, C. A. Galtieri, and B. G. Lindsay, "Transactions and Consistency in Distributed Database Management Systems," *ACM Transactions on Database Systems,* Volume 7, Number 3 (September 1982), pages 323–342.

[Tucker and Gupta 1989] A. Tucker and A. Gupta, "Process Control and Scheduling Issues for Multiprogrammed Shared-Memory Multiprocessors," *Proceedings of the Twelfth ACM Symposium on Operating System Principles* (December 1989).

[USENIX 1990] USENIX Association, *Proceedings of the Mach Workshop,* Burlington, VT (October 1990).

[USENIX 1991] USENIX Association, *Proceedings of the USENIX Mach Symposium,* Monterey, CA (November 1991).

[USENIX 1992a] USENIX Association, *Proceedings of the File Systems Workshop,* Ann Arbor, MI (May 1992).

[USENIX 1992b] USENIX Association, *Proceedings of the USENIX Workshop on Micro-kernels and Other Kernel Architectures,* Seattle, WA (April 1992).

[Vuillemin 1978] A. Vuillemin, "A Data Structure for Manipulating Priority Queues," *Communications of the ACM,* Volume 21, Number 4 (April 1978), pages 309–315.

[Wah 1984] B. W. Wah, "File Placement on Distributed Computer Systems," *Computer,* Volume 17, Number 1 (January 1984), pages 23–32.

[Walmer and Thompson 1989] L. R. Walmer and M. R. Thompson, "A Programmer's Guide to the Mach System Calls," Technical Report, Carnegie-Mellon University, Pittsburgh, PA (December 1989).

[Weizer 1981] N. Weizer, "A History of Operating Systems," *Datamation* (January 1981), pages 119–126.

[Wilhelm 1976] N. C. Wilhelm, "An Anomaly in Disk Scheduling: A Comparison of FCFS and SSTF Seek Scheduling Using an Empirical Model for Disk Accesses," *Communications of the ACM,* Volume 19, Number 1 (January 1976), pages 13–17.

[Wood and Kochan 1985] P. Wood and S. Kochan, *UNIX System Security,* Hayden, Hasbrouck Heights, NJ (1985).

[Woodside 1986] C. Woodside, "Controllability of Computer Performance Tradeoffs Obtained Using Controlled-Share Queue Schedulers," *IEEE Transactions on Software Engineering,* Volume SE-12, Number 10 (October 1986), pages 1041–1048.

[Wulf 1969] W. A. Wulf, "Performance Monitors for Multiprogramming Systems," *Proceedings of the Second ACM Symposium on Operating System Principles* (October 1969), pages 175–181.

[Wulf et al. 1981] W. A. Wulf, R. Levin, and S. P. Harbison, *Hydra/C. mmp: An Experimental Computer System,* McGraw-Hill, New York, NY (1981).

[Zahorjan and McCann 1990] J. Zahorjan and C. McCann, "Processor Scheduling in Shared Memory Multiprocessors," *Proceedings of the Conference on Measurement and Modeling of Computer Systems* (May 1990).

[Zayas 1987] E. R. Zayas, "Attacking the Process Migration Bottleneck," *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles* (October 1987), pages 13–24.

[Zhao 1989] W. Zhao, Ed., Special Issue on Real-Time Operating Systems, *Operating System Review* (July 1989).

[Zobel 1983] D. Zobel, "The Deadlock Problem: A Classifying Bibliography," *Operating Systems Review,* Volume 17, Number 4 (October 1983), pages 6–16.

CREDITS

Fig. 3.11 from Iaccobucci, *OS/2 Programmer's Guide,* © 1988, McGraw-Hill, Inc., New York, New York. Fig. 1.7, p. 20. Reprinted with permission of the publisher.

Fig. 5.8 from Khanna/Sebrée/Zolnowsky, "Realtime Scheduling in SunOS 5.0," Proceedings of Winter USENIX, January 1992, San Francisco, California. Derived with permission of the authors.

Fig. 8.28 from *80386 Programmer's Reference Manual,* Fig. 5-12, p.5-12. Reprinted by permission of Intel Corporation, Copyright/Intel Corporation 1986.

Fig. 9.15 reprinted with permission from *IBM Systems Journal,* Vol. 10, No. 3, © 1971, International Business Machines Corporation.

Fig. 11.7 from Leffler/McKusick/Karels/Quarterman, *The Design and Implementation of the 4.3BSD UNIX Operating System,* © 1989 by Addison-Wesley Publishing Co., Inc., Reading, Massachusetts. Fig. 7.6, p. 196. Reprinted with permission of the publisher.

Figs. 15.10, 15.11, and 15.13 from Halsall, *Data Communications, Computer Networks, and Open Systems, Third Edition,* © 1992, Addison-Wesley Publishing Co., Inc., Reading, Massachusetts. Fig.1.9, p. 14, fig. 1.10, p. 15, and fig. 1.11, p. 18. Reprinted with permission of the publisher.

Fig. 18.2 from Silberschatz/Korth, *Database System Concepts, Second Edition,* © 1991, McGraw-Hill, Inc., New York, New York. Fig. 15.8, p. 506. Reprinted with permission of the publisher.

Fig. 19.1 from Quarterman/Wilhelm, *UNIX, POSIX and Open Systems: The Open Standards Puzzle,* © 1993, by Addison-Wesley Publishing Co., Inc., Reading, Massachusetts. Fig. 2.1, p. 31. Reprinted with permission of the publisher.

Figs. 20.1, 20.6, and 20.8 reproduced with permission from Open Software Foundation, Inc. Excerpted from Mach Lecture Series, OSF, October 1989, Cambridge, Massachusetts.

Figs. 20.1 and 20.8 presented by R. Rashid of Carnegie Mellon University and Fig. 20.7 presented by D. Julin of Carnegie Mellon University.

Fig. 20.6 from Accetta/Baron/Bolosky/Golub/Rashid/ Tevanian/Young, "Mach: a new kernel foundation for UNIX development," Proceedings of Summer USENIX, June 1986, Atlanta, Georgia. Reprinted with permission of the authors.

Sections of chapter 6 and 18 from Silberschatz/Korth, *Database System Concepts, Second Edition,* © 1991, McGraw-Hill, Inc., New York, New York. Section 10.1.1, p. 314, 10.3.1, p. 323-324, 10.3.2, p. 329, 10.6, p. 333-334, 11.1.2, p. 352-353, 11.3.1, p. 365, 11.3.2, p. 369, 11.4.2, p. 375-376, 15.7, p. 498-500, 15.8, p. 502-506. Reprinted with permission of the publisher.

Appendix on Nachos is derived from Christopher/Procter/Anderson, "The Nachos Instructional Operating System," Proceedings of Winter USENIX, January 1993. Reprinted with permission of the authors.

Timeline information for the back end papers was assembled from a variety of sources which include "The History of Electronic Computing," compiled and edited by Marc Rettig, Association for Computing Machinery, Inc. (ACM), New York, New York, and Shedroff/Hutto/Fromm, *Understanding Computers,* © 1992, Vivid Publishing, distributed by SYBEX, San Francisco, California.

Timeline information for the front end paper was assembled by the designer of the cover, Howard S. Friedman.

INDEX

OPERATING SYSTEM CONCEPTS

Fourth Edition

Abraham Silberschatz • Peter B. Galvin

The most successful operating systems book is now in its fourth edition. This popular book enhances its reputation for clear coverage of the fundamental concepts which are the foundation of operating systems. *Operating System Concepts* has been revised to include new and updated information, examples, diagrams and an expanded bibliography.

New to this edition:

- New, early coverage of light-weight processes and threads.
- Expanded coverage of Memory Management, including modern computer architectures.
- Enhanced realistic discussion of File System design, implementation, and secondary storage management.
- Improved coverage of real time and multiprocessor systems.
- New coverage of networking as it relates to operating systems.
- Expanded and up-to-date coverage of UNIX and the Mach operating systems.
- Use of common operating systems, including Sun Solaris 2, MS-DOS, OS/2, Macintosh, in each chapter to illustrate concepts and show performance examples.

This edition retains the same high-quality of previous editions, but has expanded coverage of protection and security, and added material on atomic transactions and the two-phase commit protocol. This book is perfect for students of operating systems and for practitioners, such as system programmers, who need an understanding of the design of operating systems and enjoy having examples and projects to work with.

Abraham Silberschatz is a professor in the Department of Computer Sciences at the University of Texas at Austin, specializing in the area of concurrent processing. Professor Silberschatz is a recognized researcher, educator, and author. His research interests include operating systems, database systems, and distributed systems. Dr. Silberschatz holds an Endowed Professorship in Computer Sciences, and is a recipient of the IEEE Computer Society Outstanding Paper Award for the article "Capability Manager." He is the coauthor of *Database System Concepts.*

Peter B. Galvin is systems manager in the Department of Computer Science at Brown University. Mr. Galvin is also an independent consultant and freelance author. He has made presentations at computer conferences nationwide, and has served in leadership positions of the Sun Microsystems and Digital Equipment Corporation users groups.

Addison-Wesley Publishing Company

90000

9 780201 504804

ISBN 0-201-50480-4