UNIT - IV Deadlocks, File Systems

Deadlocks:

1. Resources
2. Conditions for resource deadlocks,
3. Ostrich algorithm,
4. Deadlock detection And recovery,
5. Deadlock avoidance,
6. Deadlock prevention.

File Systems:

1. Files
2. Directories,
3. File system implementation,
4. management and optimization.

Secondary-Storage Structure:

1. Overview of disk structure,
2. and attachment,
3. Disk scheduling,
4. RAID structure,
5. Stable storage implementation.

Chapter-1

DEADLOCKS

1)RESOURCES

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

Preemptable and Nonpreemptable Resources

Resources come in two types**: preemptable and nonpreemptable**. A preemptable resource is one that can be taken away from the process owning it with no ill effects. **Memory is an example of a preemptable resource**. Consider, for example, a system with 1 GB of user memory, one printer, and two 1-GB processes that each want to print something. Process A requests and gets the printer, then starts to compute the values to print. Before it has finished the computation, it exceeds its time quantum and is swapped out to disk. Process B now runs and tries, unsuccessfully as it turns out, to acquire the printer. Potenially, we now hav e a deadlock situation, because A has the printer and B has the memory, and neither one can proceed

without the resource held by the other. Fortunately, it is possible to preempt (take away) the memory from B by

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

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

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

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

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

1. Request the resource.

2. Use the resource.

3. Release the resource.

If the resource is not available when it is requested, the requesting process is forced 0 to wait. In some operating systems, **the process is automatically blocked when a resource request fails**, and awakened when it becomes available. In other systems,the request fails with an error code, and it is up to the calling process to wait a little while and try again.

A process whose resource request has just been denied will normally sit in a tight loop requesting the resource, then sleeping, then trying again. Although this process is not blocked, for all intents and purposes it is as good as blocked, because it cannot do any useful work. In our further treatment, we will assume that when a process is denied a resource request, it is put to sleep.The exact nature of requesting a resource is highly system dependent. In some systems, a request system call is provided to allow processes to explicitly ask for resources. In others, the only resources that the operating system knows about are special files that only one process can have open at a time. These are opened by the usual open call. If the file is already in use, the caller is blocked until its current owner closes it.

2)Conditions for Resource Deadlocks

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

(resource) deadlock:

1. Mutual exclusion condition. Each resource is either currently assigned to exactly one process or is available.

2. Hold-and-wait condition. Processes currently holding resources that were granted earlier can request new resources.

3. No-preemption condition. Resources previously granted cannot be forcibly taken away from a process. They must be explicitly released by the process holding them.

4. Circular wait condition. There must be a circular list of two or more processes, each of which is waiting for a resource held by the next member of the chain.

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

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

Deadlock Modeling

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

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

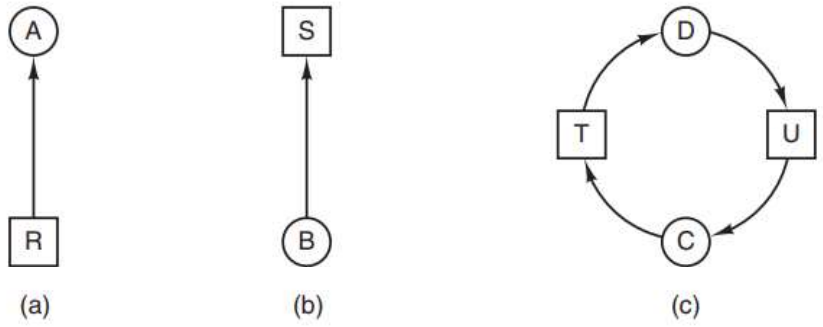

Figure 6-3. Resource allocation graphs. (a) Holding a resource. (b) Requesting a resource. (c) Deadlock.

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

1. Just ignore the problem. Maybe if you ignore it, it will ignore you.
2. Detection and recovery. Let them occur, detect them, and take action.
3. Dynamic avoidance by careful resource allocation.
4. Prevention, by structurally negating one of the four conditions.
In the next four sections, we will examine each of these methods in turn.

3)THE OSTRICH ALGORITHM

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

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

4).DEADLOCK DETECTION AND RECOVERY

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

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

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

Deadlock Detection with One Resource of Each Type

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

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

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

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

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

. The cycle is shown in Fig. 6-5(b). From this cycle**, we can see that processes D, E, and G are all deadlocked**. Processes A, C, and F are not deadlocked because S can be allocated to any one of them, which then finishes and returns it. Then the other two can take it in turn and also complete

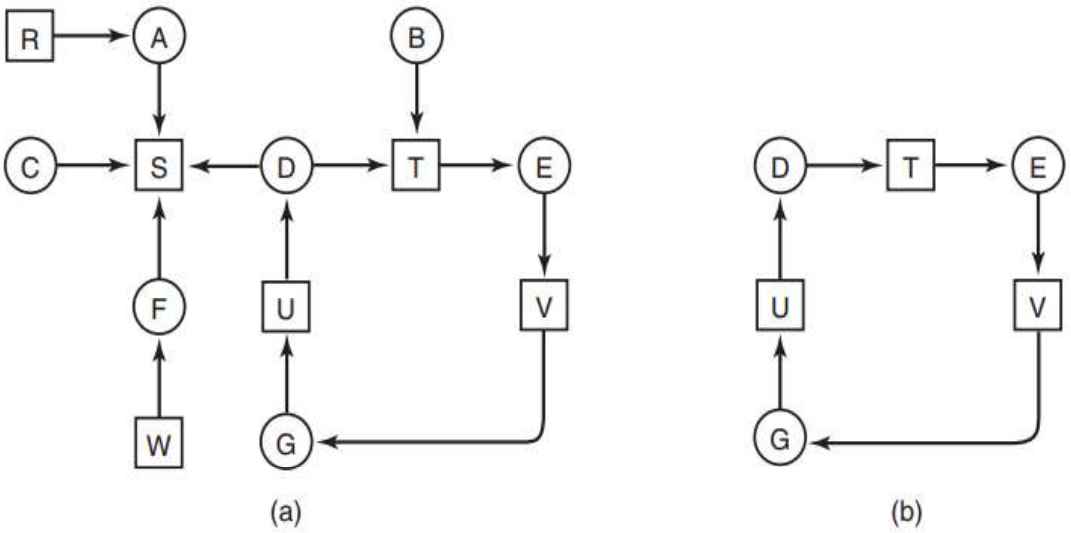

Figure 6-5. (a) A resource graph. (b) A cycle extracted from (a).

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

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

1. For each node, N, in the graph, perform the following fiv e steps with N as the starting node.

2. Initialize L to the empty list, and designate all the arcs as unmarked.

3. Add the current node to the end of L and check to see if the node now appears in L two times. If it does, the graph contains a cycle (listed in

L) and the algorithm terminates.

4. From the given node, see if there are any unmarked outgoing arcs. If so, go to step 5; if not, go to step 6.

5. Pick an unmarked outgoing arc at random and mark it. Then follow it to the new current node and go to step 3.

6. If this node is the initial node, the graph does not contain any cycles and the algorithm terminates.

Otherwise, we have now reached a dead end. Remove it and go back to the previous node, that is, the one that was current just before this one, make that one the current node, and go to step 3.

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

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

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

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

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

Deadlock Detection with Multiple Resources of Each Type

When multiple copies of some of the resources exist, a different approach is needed to detect deadlocks**. We will now present a matrix-based algorithm for detecting deadlock among n processes**, P1 through Pn. Let the number of resource classes be m, with E1 resources of class 1, E2 resources of class 2, and generally, Ei resources of class i ($1 \le i \le m$). **E is the existing resource vector.** It giv es the total number of instances of each resource in existence. For example, if class 1 is tape drives, then E1 = 2 means the system has two tape drives.

At any instant, some of the resources are assigned and are not available**. Let A be the available resource vector**, with Ai giving the number of instances of resource i that are currently available (i.e., unassigned). If both of our two tape drives are assigned, A1 will be 0.

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

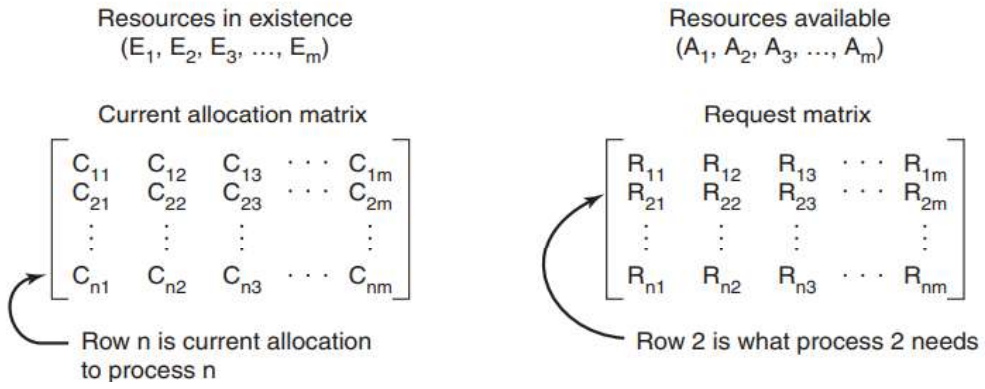

Figure 6-6. The four data structures needed by the deadlock detection algorithm.

- The deadlock detection algorithm is based on comparing vectors. **Let us define the relation A ≤ B on two vectors A and B to mean that each element of A is less than or equal to the corresponding element of B.** Mathematically,A ≤ B holds **if and only if Ai ≤ Bi for 1 ≤ i ≤ m.**

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

The deadlock detection algorithm can now be given as follows:

1. Look for an unmarked process, Pi, for which the ith row of **R is less than or equal to A.**

2. If such a process is found, add the ith row of C to A, mark the process, and go back to step 1.

3. If no such process exists, **the algorithm terminates**.

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

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

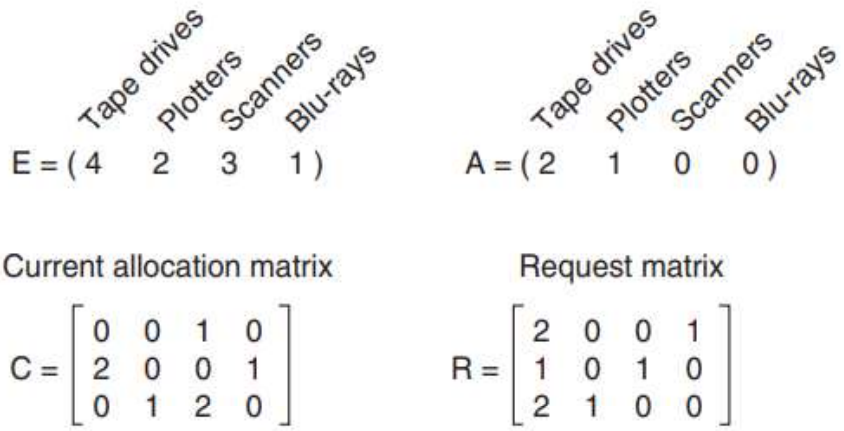

Figure 6-7. An example for the deadlock detection algorithm.

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

A = (2 2 2 0)

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

A = (4 2 2 1)

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

Recovery from Deadlock

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

Recovery through Preemption :

In some cases it may be possible to temporarily take a resource away from its current owner and give it to another process. **In many cases, manual intervention may be required, especially in batch-processing operating systems running on mainframes.**
For example, to take a laser printer away from its owner, the operator can collect all the sheets already printed and put them in a pile. Then the process can be suspended (marked as not runnable). At this point the printer can be assigned to another process. When that process finishes, the pile of printed sheets can be put back in the printer's output tray and the original process restarted.
The ability to take a resource away from a process, have another process use it, and then give it back without the process noticing it is highly dependent on the nature of the resource. Recovering this way is frequently difficult or impossible. Choosing the process to suspend depends largely on which ones have resources that can easily be taken back.

Recovery through Rollback

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

Recovery through Killing Processes:

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

Where possible, it is best to kill a process that can be rerun from the beginning with no ill effects. For example, a compilation can always be rerun because all it does is read a source file and produce an object file. If it is killed partway through, the first run has no influence on the second run.
On the other hand, a process that updates a database cannot always be run a second time safely. If the process adds 1 to some field of a table in the database, running it once, killing it, and then running it again will add 2 to the field, which is incorrect.

DEADLOCK AV OIDANCE :

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

Resource Trajectories:

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

In Fig. 6-8 we see a model for dealing with two processes and two resources, for example, a printer and a plotter. The horizontal axis represents the number of instructions executed by process A. The vertical axis represents the number of instructions executed by process B. At I1 A requests a printer; at I2 it needs a plotter. The printer and plotter are released at I3 and I4, respectively. Process B needs the plotter from I5 to I7 and the printer from I6 to I8.

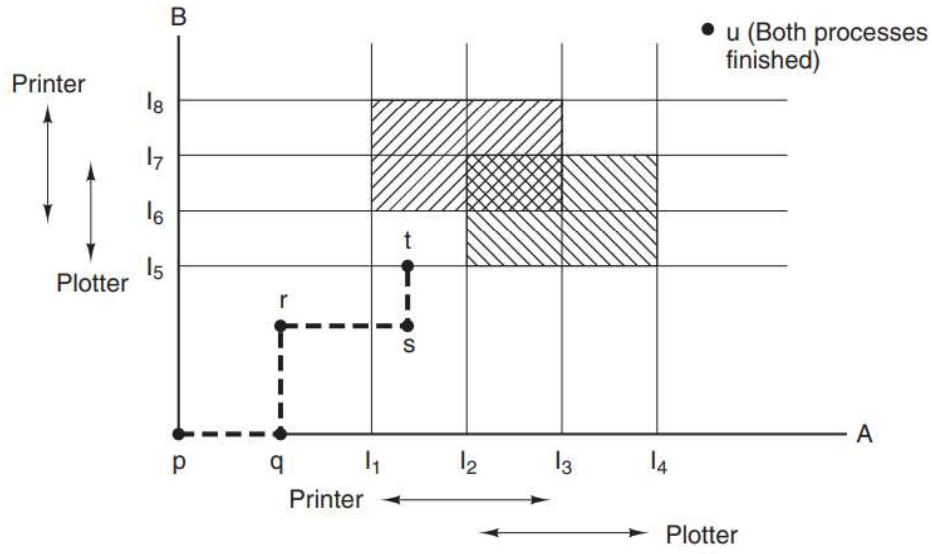

Figure 6-8. Two process resource trajectories.

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

When A crosses the I1 line on the path from r to s, it requests and is granted the printer. When B reaches point t, it requests the plotter. The regions that are shaded are especially interesting. The region with lines slanting from southwest to northeast represents both processes having the printer. The mutual exclusion rule makes it impossible to enter this region. Similarly, the region shaded the other way represents both processes having the plotter and is equally impossible. If the system ever enters the box bounded by I1 and I2 on the sides and I5 and I6 top and bottom, it will eventually deadlock when it gets to the intersection of I2 and I6. At this point, A is requesting the plotter and B is requesting the printer, and both are already assigned. The entire box is unsafe and must not be entered.

Safe and Unsafe States

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

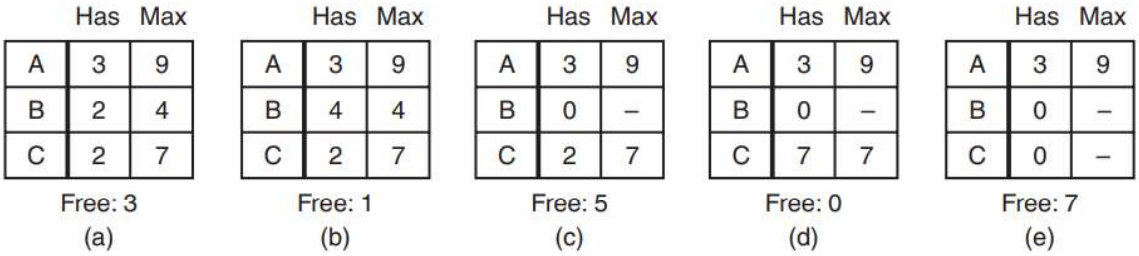

Figure 6-9. Demonstration that the state in (a) is safe.

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

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

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

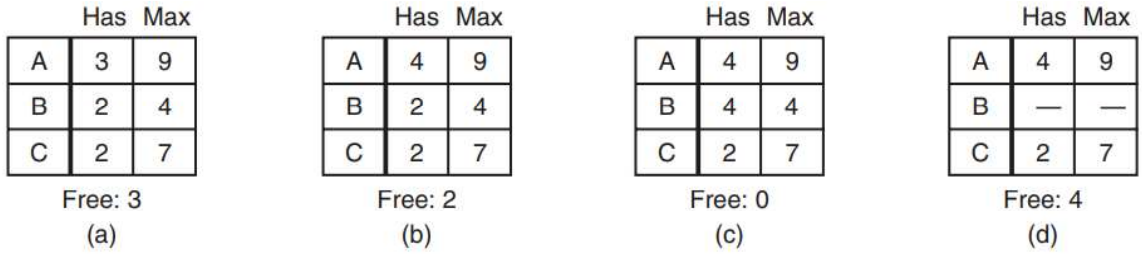

Figure 6-10. Demonstration that the state in (b) is not safe.

processes needs fiv e. There is no sequence that guarantees completion. Thus, the allocation decision that moved the system from Fig. 6-10(a) to Fig. 6-10(b) went from a safe to an unsafe state. Running A or C next starting at Fig. 6-10(b) does not work either. In retrospect, A's request should not have been granted. It is worth noting that an unsafe state is not a deadlocked state. Starting at Fig. 6-10(b), the system can run for a while. In fact, one process can even complete. Furthermore, it is possible that A might release a resource before asking for any more, allowing C to complete and avoiding deadlock altogether. Thus**, the difference between a safe state and an unsafe state is that from a safe state the system can guarantee that all processes will finish; from an unsafe state, no such guarantee can be give**n.

The Banker's Algorithm for a Single Resource

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

than 22 to service them. (In this analogy, customers are processes, units are, say, tape drives, and the banker is the operating system.)

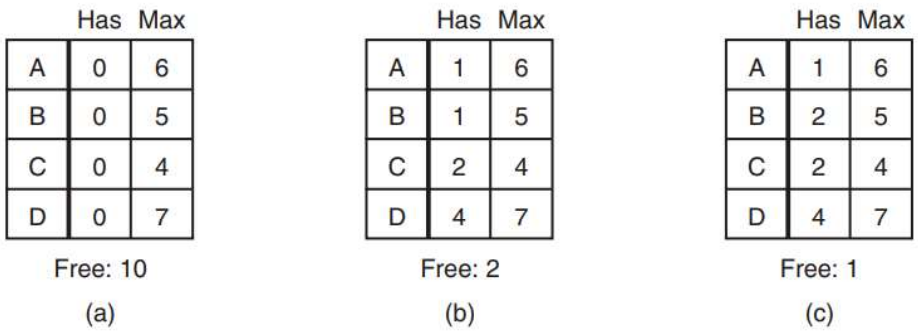

Figure 6-11. Three resource allocation states: (a) Safe. (b) Safe. (c) Unsafe.

the customers suddenly asked for their maximum loans, the banker could not satisfy any of them, and we would have a deadlock. An unsafe state does not have to lead to deadlock**, since a customer might not need the entire credit line available, but the banker cannot count on this behavior**.
The banker's algorithm considers each request as it occurs, seeing whether granting it leads to a safe state. If it does, the request is granted; otherwise, it is postponed until later. To see if a state is safe, the banker checks to see if he has enough resources to satisfy some customer. If so, those loans are assumed to be repaid, and the customer now closest to the limit is checked, and so on. If all loans can eventually be repaid, the state is safe and the initial request can be granted.

The Banker's Algorithm for Multiple Resources

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

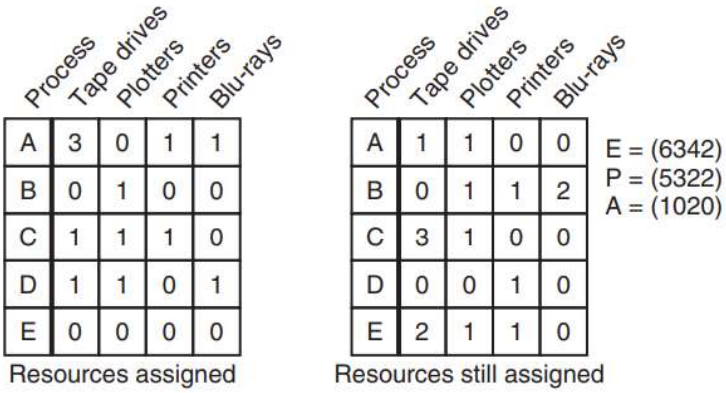

Figure 6-12. The banker's algorithm with multiple resources.

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

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

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

1. Look for a row, R, whose unmet resource needs are all smaller than or equal to A. If no such row exists, the system will eventually deadlock since no process can run to completion (assuming processes keep all resources until they exit).

2. Assume the process of the chosen row requests all the resources it needs (which is guaranteed to be possible) and finishes. Mark that process as terminated and add all of its resources to the A vector.

3. Repeat steps 1 and 2 until either all processes are marked terminated (in which case the initial state was safe) or no process is left whose resource needs can be met (in which case the system was not safe).
If several processes are eligible to be chosen in step 1, it does not matter which one is selected: the pool of available resources either gets larger, or at worst, stays the same.
the banker's algorithm to prevent deadlock. For instance, networks may throttle traffic when buffer utilization reaches higher than, say, 70%—estimating that the remaining 30% will be sufficient for current users to complete their service and return their resources.

DEADLOCK PREVENTION

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

Attacking the Mutual-Exclusion Condition

First let us attack the mutual exclusion condition. If no resource were ever assigned exclusively to a single process, we would never have deadlocks. For data, the simplest method is to make data read only, so that processes can use the data **concurrently**. However, it is equally clear that allowing two processes to write on the printer at the same time will lead to chaos. **By spooling printer output, several processes can generate output at the same time.** In this model, the only process that actually requests the physical printer is the printer daemon. Since the daemon never requests any other resources, we can eliminate deadlock for the printer.
If **the daemon is programmed to begin printing even before all the output is spooled**, the printer might lie idle if an output process decides to wait several hours after the first burst of output. For this reason, daemons are normally programmed to print only after the complete output file is available. However, this decision itself could lead to deadlock. What would happen if two processes each filled up one half of the available spooling space with output and neither was finished producing its full output? In this case, we would have two processes that had each finished part, but not all, of their output, and could not continue. Neither process will ev er finish, so we would have a deadlock on the disk. **Nevertheles**s, there is a germ of an idea here that is frequently applicable. Avoid assigning a resource unless absolutely necessary, and try to make sure that as few processes as possible may actually claim the resource.

Attacking the Hold-and-Wait Condition

The second of the conditions stated by Coffman et al. looks slightly more promising. If we can prevent processes that hold resources from waiting for more resources, we can eliminate deadlocks. One way to achieve this goal is to require12 all processes to request all their resources before starting execution. If ev erything is available, the process will be allocated whatever it needs and can run to completion. If one or more resources are busy, nothing will be allocated and the process will just wait.

An immediate problem with this approach is that many processes do not know how many resources they will need until they hav e started running. In fact, if they knew, the banker's algorithm could be used. Another problem is that resources will not be used optimally with this approach. Take, as an example, a

process that reads data from an input tape, analyzes it for an hour, and then writes an output tape as well as plotting the results. If all resources must be requested in advance, the process will tie up the output tape drive and the plotter for an hour.

Attacking the No-Preemption Condition

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

However, not all resources can be virtualized like this. For example, records in databases or tables inside the operating system must be locked to be used and therein lies the potential for deadlock..

Attacking the Circular Wait Condition

Only one condition is left. The circular wait can be eliminated in several ways. One way is simply to have a rule saying that a process is entitled only to a single resource at any moment. If it needs a second one, it must release the first one. For a process that needs to copy a huge file from a tape to a printer, this restriction is unacceptable.

Another way to avoid the circular wait is to provide a global numbering of all the resources, as shown in Fig. 6-13(a). Now the rule is this: processes can request

resources whenever they want to, but all requests must be made in numerical order. A process may request first a printer and then a tape drive, but it may not request first a plotter and then a printer.

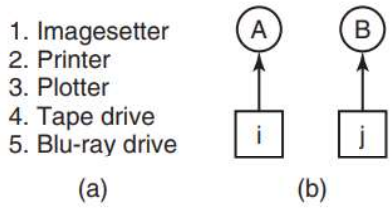

Figure 6-13. (a) Numerically ordered resources. (b) A resource graph.

With this rule, the resource allocation graph can never have cycles. Let us see why this is true for the case of two processes, in Fig. 6-13(b). We can get a deadlock only if A requests resource j and B requests resource i. Assuming i and j are distinct resources, they will have different numbers. If $i > j$, then A is not allowed to request j because that is lower than what it already has. If $i < j$, then B is not allowed to request i because that is lower than what it already has. Either way, deadlock is impossible.

OTHER ISSUES

In this section we will discuss a few miscellaneous issues related to deadlocks**. These include two-phase locking, nonresource deadlocks, and starvation**

Condition	Approach
Mutual exclusion	Spool everything
Hold and wait	Request all resources initially
No preemption	Take resources away
Circular wait	Order resources numerically

Figure 6-14. Summary of approaches to deadlock prevention.

Two-Phase Locking

Although both avoidance and prevention are not terribly promising in the general case, for specific applications, many excellent special-purpose algorithms are known. As an example, in many database systems, an operation that occurs frequently is requesting locks on several records and then updating all the locked records. When multiple processes are running at the same time, there is a real danger of deadlock.

The approach often used is called **two-phase locking**. In the first phase, the process tries to lock all the records it needs, one at a time. If it succeeds, it begins the second phase, performing its updates and releasing the locks. No real work is done in the first phase.

If during the first phase, some record is needed that is already locked, the process just releases all its locks and starts the first phase all over. In a certain sense, this approach is similar to requesting all the resources needed in advance, or at least before anything irreversible is done. In some versions of two-phase locking, there is no release and restart if a locked record is encountered during the first phase. In these versions, deadlock can occur.

However, this strategy is not applicable in general. In real-time systems and process control systems, for example, it is not acceptable to just terminate a process partway through because a resource is not available and start all over again. Neither is it acceptable to start over if the process has read or written messages to the network, updated files, or anything else that cannot be safely repeated. The algorithm works only in those situations where the programmer has very carefully arranged things so that the program can be stopped at any point during the first phase and restarted. Many applications cannot be structured this way.

Communication Deadlocks

All of our work so far has concentrated on resource deadlocks. One process wants something that another process has and must wait until the first one gives it up. Sometimes the resources are hardware or software objects, such as Blu-ray drives or database records, but sometimes they are more abstract. Resource deadlock is a problem of competition synchronization. Independent processes complete service if their execution were not interleaved with competing processes

This, though, is not the classical resource deadlock. A does not have possession of some resource B wants, and vice versa. In fact, there are no resources at all in sight. But it is a deadlock according to our formal definition since we have a set of (two) processes, each blocked waiting for an event only the other one can cause. This situation is called a communication deadlock to contrast it with the more common resource deadlock. Communication deadlock is an anomaly of cooperation synchronization. The processes in this type of deadlock could not complete service if executed independently.

Communication deadlocks cannot be prevented by ordering the resources (since there are no resources) or avoided by careful scheduling (since there are no moments when a request could be postponed). Fortunately, there is another technique that can usually be employed to break communication deadlocks: timeouts. In most network communication systems, whenever a message is sent to which a reply is expected, a timer is started. If the timer goes off before the reply arrives, the sender of the message assumes that the message has been lost and sends it again (and again and again if needed). In this way, the deadlock is broken. Phrased differently, the timeout serves as a heuristic to detect deadlocks and enables recovery. This heuristic is applicable to resource deadlock also and is relied upon by users with temperamental or buggy device drivers that can deadlock and freeze the system.

Not all deadlocks occurring in communication systems or networks are communication deadlocks. Resource deadlocks can also occur there. Consider, for example, the network of Fig. 6-15. It is a simplified view of the Internet. Very simplified. The Internet consists of two kinds of computers: hosts and routers. A host is a user computer, either someone's tablet or PC at home, a PC at a company, or a corporate server. Hosts do work for people. A router is a specialized communications computer that moves packets of data from the source to the destination. Each host is connected to one or more routers, either by a DSL line, cable TV connection, LAN, dial-up line, wireless network, optical fiber, or something else.

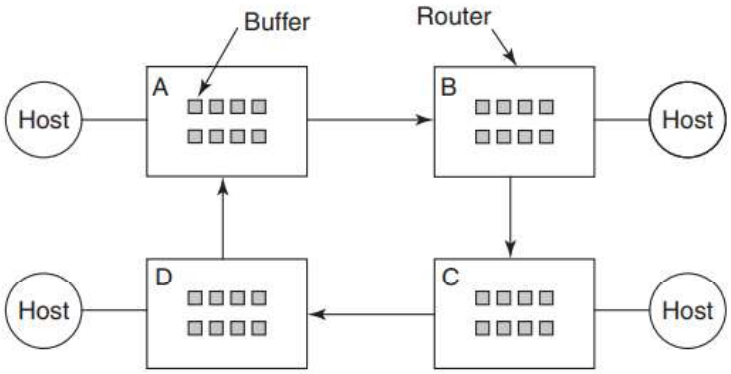

Figure 6-15. A resource deadlock in a network.

When a packet comes into a router from one of its hosts, it is put into a buffer for subsequent transmission to another router and then to another until it gets to the destination. These buffers are resources and there are a finite number of them. In Fig. 6-16 each router has only eight buffers (in practice they hav e millions, but that does not change the nature of the potential deadlock, just its frequency). Suppose that all the packets at router A need to go to B and all the packets at B need to go to C and all the packets at C need to go to D and all the packets at D need to go to A. No packet can move because there is no buffer at the other end and we have a classical resource deadlock, albeit in the middle of a communications system.

Starvation:

A problem closely related to deadlock and livelock is **starvation**. In a dynamic system, requests for resources happen all the time. Some policy is needed to make a decision about who gets which resource when. This policy, although seemingly reasonable, may lead to some processes never getting service even though they are not deadlocked.

As an example, consider allocation of the printer. Imagine that the system uses some algorithm to ensure that allocating the printer does not lead to deadlock. Now suppose that several processes all want it at once. Who should get it?

One possible allocation algorithm is to give it to the process with the smallest file to print (assuming this information is available). This approach maximizes the number of happy customers and seems fair. Now consider what happens in a busy system when one process has a huge file to print. Every time the printer is free, the system will look around and choose the process with the shortest file. If there is a constant stream of processes with short files, the process with the huge file will never be allocated the printer. It will simply starve to death (be postponed indefinitely, even though it is not blocked).

Starvation can be avoided by using a first-come, first-served resource allocation policy. With this approach, the process waiting the longest gets served next. In due course of time, any given process will eventually become the oldest and thus get the needed resource.

CHAPTER-2

File Systems:

1. Files
2. Directories,
3. File system implementation,
4. management and optimization.

1)Files:

All computer applications need to store and retrieve information. While a process is running, it can store a limited amount of information within its own address space. However, the storage capacity is restricted to the size of the virtual address space. For some applications this size is adequate, but for others, such as airline reservations, banking, or corporate record keeping, it is far too small.

A second problem with keeping information within a process' address space is that when the process terminates, the information is lost. For many applications (e.g., for databases), the information must be retained for weeks, months, or even forever. Having it vanish when the process using it terminates is unacceptable. Furthermore, it must not go away when a computer crash kills the process.

A third problem is that it is frequently necessary for multiple processes to access (parts of) the information at the same time. If we have an online telephone directory stored inside the address space of a single process, only that process can access it. The way to solve this problem is to make the information itself independent of any one process.

Thus, we have three essential requirements for long-term information storage:

1. It must be possible to store a very large amount of information.
2. The information must survive the termination of the process using it.
3. Multiple processes must be able to access the information at once.

Magnetic disks have been used for years for this long-term storage. In recent years, solid-state drives have become increasingly popular, as they do not have any

moving parts that may break. Also, they offer fast random access. Tapes and optical disks have also been used extensively, but they hav e much lower performance and are typically used for backups. We will study disks more in Chap. 5, but for the moment, it is sufficient to think of a disk as a linear sequence of fixed-size blocks and supporting two operations:

1. Read block k.

2. Write block k

In reality there are more, but with these two operations one could, in principle, solve the long-term storage problem. However, these are very inconvenient operations, especially on large systems used by many applications and possibly multiple users (e.g., on a server). Just a few of the questions that quickly arise are:

1. How do you find information?

2. How do you keep one user from reading another user's data?

3. How do you know which blocks are free?

Files are logical units of information created by processes. A disk will usually contain thousands or even millions of them, each one independent of the others. In fact, if you think of each file as a kind of address space, you are not that far off, except that they are used to model the disk instead of modeling the RAM. ***Processes can read existing files and create new ones if need be. Information stored in files must be persistent, that is, not be affected by process creation and termination***.

File Naming:

An aside on file systems is probably in order here. Windows 95 and Windows 98 both used the MS-DOS file system, called FAT-16, and thus inherit many of its properties, such as how file names are constructed. Windows 98 introduced some extensions to FAT -16, leading to FAT-32, but these two are quite similar. In addition, Windows NT, Windows 2000, Windows XP, Windows Vista, Windows 7, and Windows 8 all still support both FAT file systems, which are really obsolete now. However, these newer operating systems also

have a much more advanced native file system (NTFS) that has different properties (such as file names in Unicode).

Many operating systems support two-part file names, with the two parts separated by a period, as in prog.c. The part following the period is called the file extension and usually indicates something about the file. In MS-DOS, for example, file names are 1 to 8 characters, plus an optional extension of 1 to 3 characters. In UNIX, the size of the extension, if any, is up to the user, and a file may ev en hav e two or more extensions, as in homepage.html.zip, where .html indicates a Web page in HTML and .zip indicates that the file (homepage.html) has been compressed using the zip program. Some of the more common file extensions and their meanings are shown in Fig. 4-1.

Extension	Meaning
.bak	Backup file
.c	C source program
.gif	Compuserve Graphical Interchange Format image
.hlp	Help file
.html	World Wide Web HyperText Markup Language document
.jpg	Still picture encoded with the JPEG standard
.mp3	Music encoded in MPEG layer 3 audio format
.mpg	Movie encoded with the MPEG standard
.o	Object file (compiler output, not yet linked)
.pdf	Portable Document Format file
.ps	PostScript file
.tex	Input for the TEX formatting program
.txt	General text file
.zip	Compressed archive

Figure 4-1. Some typical file extensions.

In contrast, Windows is aware of the extensions and assigns meaning to them. Users (or processes) can register extensions with the operating system and specify for each one which program **''owns''** that extension. When a user double clicks on a file name, the program assigned to its file extension is launched with the file as parameter. For example, double clicking on **file.docx** starts Microsoft Word with file.docx as the initial file to edit.

File Structure:

Files can be structured in any of sev eral ways. Three common possibilities are depicted in Fig. 4-2. The file in Fig. 4-2(a) is an unstructured sequence of bytes. In effect, the operating system does not know or care what is in the file. All it sees are bytes. Any meaning must be imposed by user-level programs. Both UNIX and Windows use this approach.

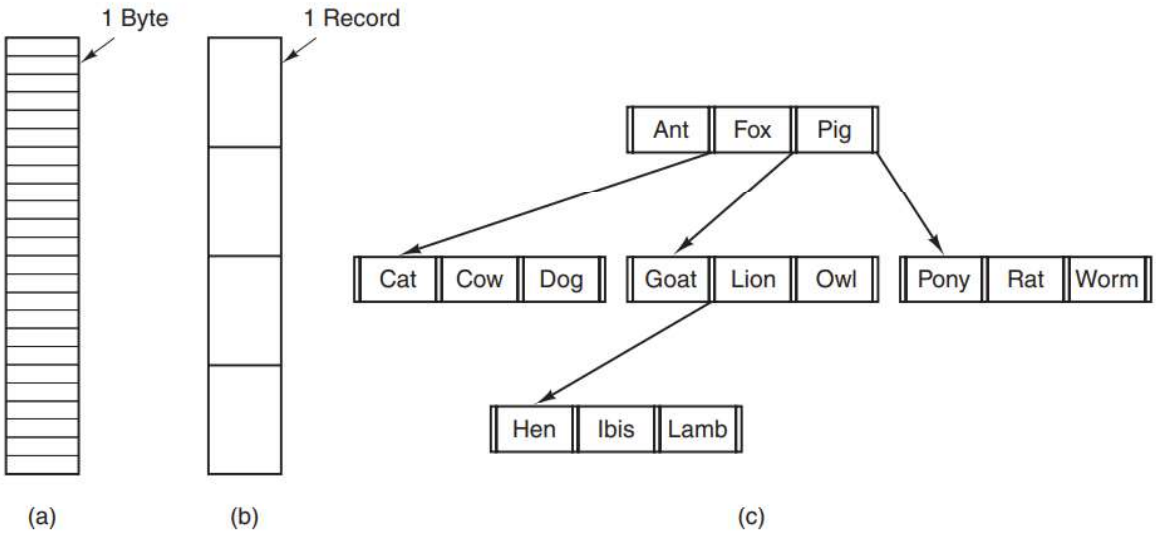

Figure 4-2. Three kinds of files. (a) Byte sequence. (b) Record sequence. (c) Tree.

Having the operating system regard files as nothing more than byte sequences provides the maximum amount of flexibility. **User programs can put anything they want in their files and name them any way that they find convenient.** The operating system does not help, but it also does not get in the way. For users who want to do unusual things, the latter can be very important. All versions of UNIX (including Linux and OS X) and Windows use this file model.

The first step up in structure isillustrated in Fig. 4-2(b). In this model, **a file is a sequence of fixed-length records, each with some internal structure**. Central to the idea of a file being a sequence of records is the idea that the read operation returns one record and the write operation overwrites or appends one record. As a historical note, in decades gone by, when the 80-column punched card was king of the mountain, many (mainframe) operating systems based their file systems on files consisting **of 80-character records**, in effect, card images. **These systems also supported files of 132-character records**, which were intended for the line printer (which in those days were big chain printers having 132 columns). Programs read input in units of 80 characters and wrote it in units of 132 characters, although the final **52 could be spaces**, of course. No current general-purpose system uses this model as its primary file system any more, but back in the days of 80-column punched cards and 132-character line printer paper this was a common model on mainframe computers.

The third kind of file structure is shown in Fig. 4-2(c). In this organization, a file consists of a tree of records, not necessarily all the same length, each containing a **key field** in a fixed position in the record. **The tree is sorted on the key field, to allow rapid searching for a particular key.**

File Types:

Many operating systems support several types of files. UNIX (again, including OS X) and Windows, for example, have regular files and directories. UNIX also has character and block special files. Regular files are the ones that contain user information. All the files of Fig. 4-2 are regular files. Directories are system files for maintaining the structure of the file system. We will study directories **below. Character special files are related to input/output and used to model serial I/O devices, such as terminals, printers, and networks. Block special files are used to model disks.** In this chapter we will be primarily interested in regular files. Regular files are generally either ASCII files or binary files. ASCII files consist of lines of text. In some systems each line is terminated by a carriage return character. In others, the line feed character is used. Some systems (e.g., Windows) use both. Lines need not all be of the same length.

For example, in Fig. 4-3(a) we see a simple executable binary file taken from an early version of UNIX. Although technically the file is just a sequence of bytes, the operating system will execute a file only if it has

the proper format. It has five sections: header, text, data, relocation bits, and symbol table. The header starts with a so-called magic number, identifying the file as an executable file (to prevent the accidental execution of a file not in this format). Then come the sizes of the various pieces of the file, the address at which execution starts, and some flag bits. Following the header are the text and data of the program itself. These are loaded into memory and relocated using the relocation bits. The symbol table is used for debugging.

s. Copying them to the printer would produce complete gibberish. Every operating system must recognize at least one file type: its own executable file; some recognize more. The old TOPS-20 system (for the DECsystem 20) went so far as to examine the creation time of any file to be executed. Then it located the source file and saw whether the source had been modified since the binary was made. If it had been, it automatically recompiled the source. In UNIX terms, the make program had been built into the shell. The file extensions were mandatory, so it could tell which binary program was derived from which source.

Having strongly typed files like this causes problems whenever the user does anything that the system designers did not expect. Consider, as an example, a system in which program output files have extension .dat (data files). If a user writes a program formatter that reads a .c file (C program), transforms it (e.g., by converting it to a standard indentation layout), and then writes the transformed file as output, the output file will be of type .dat. If the user tries to offer this to the C compiler to compile it, the system will refuse because it has the wrong extension. Attempts to copy file.dat to file.c will be rejected by the system as invalid (to protect the user against mistakes)

While this kind of ‘‘**user friendliness’’** may help novices, it drives experienced users up the wall since they hav e to devote considerable effort to circumventing the operating system’s idea of what is reasonable and what is not.

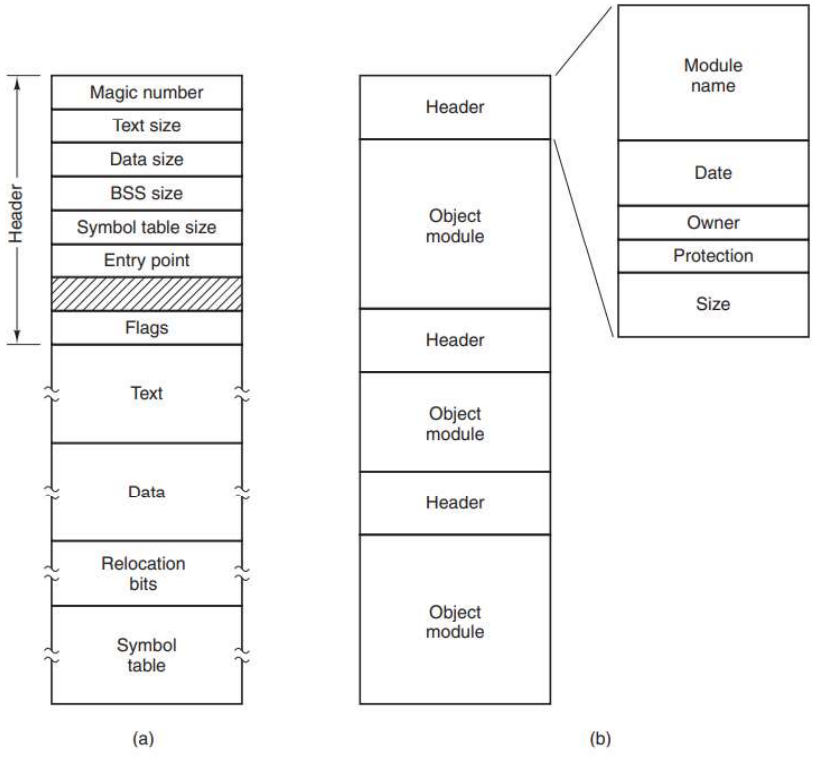

Figure 4-3. (a) An executable file. (b) An archive.

File Access:

Early operating systems provided only one kind of file access*: sequential access. In these systems, a process could read all the bytes or records in a file in order, starting at the beginning, but could not skip around and read them out of order.* Sequential files could be rewound, however, so they could be read as often as needed. **Sequential files were convenient when the storage medium was magnetic tape** rather than disk.

When disks came into use for storing files, it became possible to read the bytes or records of a file out of order, or to access records by key rather than by position. **Files whose bytes or records can be read in any order are called random-access files. They are required by many application**

Random access files are essential for many applications, for example, ***database systems.*** **If an airline customer calls up and wants to reserve a seat on a particular flight**, the reservation program must be able to access the record for that flight without having to read the records for thousands of other flights first

File Attributes:

Every file has a name and its data. In addition, all operating systems associate other information with each file, for example, the date and time the file was last modified and the file's size. We will call these extra items the **file's attributes. Some people call them metadata**. The list of attributes varies considerably from system to system. The table of Fig. 4-4 shows some of the possibilities, but other ones also exist. No existing system has all of these, but each one is present in some system**.**

The current size tells how big the file is at present. Some old mainframe operating systems required the maximum size to be specified when the file was created, in order to let the operating system reserve the maximum amount of storage in advance. Workstation and personal-computer operating systems are thankfully clever enough to do without this feature nowadays.

Attribute	Meaning
Protection	Who can access the file and in what way
Password	Password needed to access the file
Creator	ID of the person who created the file
Owner	Current owner
Read-only flag	0 for read/write; 1 for read only
Hidden flag	0 for normal; 1 for do not display in listings
System flag	0 for normal files; 1 for system file
Archive flag	0 for has been backed up; 1 for needs to be backed up
ASCII/binary flag	0 for ASCII file; 1 for binary file
Random access flag	0 for sequential access only; 1 for random access
Temporary flag	0 for normal; 1 for delete file on process exit
Lock flags	0 for unlocked; nonzero for locked
Record length	Number of bytes in a record
Key position	Offset of the key within each record
Key length	Number of bytes in the key field
Creation time	Date and time the file was created
Time of last access	Date and time the file was last accessed
Time of last change	Date and time the file was last changed
Current size	Number of bytes in the file
Maximum size	Number of bytes the file may grow to

Figure 4-4. Some possible file attributes.

File Operations

Files exist to store information and allow it to be retrieved later. Different systems provide different operations to allow storage and retrieval. Below is a discussion of the most common system calls relating to files.

1. **Create**. The file is created with no data. The purpose of the call is to announce that the file is coming and to set some of the attributes.

2. **Delete**. When the file is no longer needed, it has to be deleted to free up disk space. There is always a system call for this purpose

3. **Open**. Before using a file, a process must open it. The purpose of the open call is to allow the system to fetch the attributes and list of disk addresses into main memory for rapid access on later calls.

4**. Close**. When all the accesses are finished, the attributes and disk addresses are no longer needed, so the file should be closed to free up internal table space. Many systems encourage this by imposing a SEC. 4.1 FILES 273 maximum number of open files on processes. A disk is written in blocks, and closing a file forces writing of the file's last block, even though that block may not be entirely full yet.

5. **Read.** Data are read from file. Usually, the bytes come from the current position. The caller must specify how many data are needed and must also provide a buffer to put them in.

6. **Write**. Data are written to the file again, usually at the current position. If the current position is the end of the file, the file's size increases. If the current position is in the middle of the file, existing data are overwritten and lost forever.

7. **Append**. This call is a restricted form of wr ite. It can add data only to the end of the file. Systems that provide a minimal set of system calls rarely have append, but many systems provide multiple ways of doing the same thing, and these systems sometimes have append.

8. **Seek.** For random-access files, a method is needed to specify from where to take the data. One common approach is a system call, seek, that repositions the file pointer to a specific place in the file. After this call has completed, data can be read from, or written to, that position.

9. **Get attributes**. Processes often need to read file attributes to do their work. For example, the UNIX make program is commonly used to manage software development projects consisting of many source files. When make is called, it examines the modification times of all the source and object files and arranges for the minimum number of compilations required to bring everything up to date. To do its job, it must look at the attributes, namely, the modification times.

10. **Set attributes**. Some of the attributes are user settable and can be changed after the file has been created. This system call makes that possible. The protection-mode information is an obvious example. Most of the flags also fall in this category.

11. **Renam**e. It frequently happens that a user needs to change the name of an existing file. This system call makes that possible. It is not always strictly necessary, because the file can usually be copied to a new file with the new name, and the old file then deleted.

2)DIRECTORIES :

To keep track of files, file systems normally have directories or folders, which are themselves files. In this section we will discuss directories, their organization, their properties, and the operations that can be performed on them.

Single-Level Directory Systems :

The simplest form of directory system is having one directory containing all the files. Sometimes it is called the root directory, but since it is the only one, the name does not matter much. On early personal computers, this system was common, in part because there was only one user. Interestingly enough, the world's first supercomputer, the CDC 6600, also had only a single directory for all files, even though it was used by many users at once. This decision was no doubt made to keep the software design simple. An example of a system with one directory is given in Fig. 4-6. Here the directory contains four files. The

advantages of this scheme are its simplicity and the ability to locate files quickly—there is only one place to look, after all. It is sometimes still used on simple embedded devices such as digital cameras and some portable music players.

Hierarchical Directory Systems:

The single level is adequate for very simple dedicated applications (and was even used on the first personal computers), but for modern users with thousands of files, it would be impossible to find anything if all files were in a single directory.

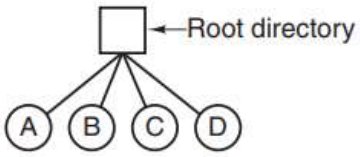

Figure 4-6. A single-level directory system containing four files.

Consequently, a way is needed to group related files together. A professor, for example, might have a collection of files that together form a book that he is writing, a second collection containing student programs submitted for another course, a third group containing the code of an advanced compiler-writing system he is building, a fourth group containing grant proposals, as well as other files for electronic mail, minutes of meetings, papers he is writing, games, and so on.

What is needed is a hierarchy (i.e., a tree of directories). With this approach, there can be as many directories as are needed to group the files in natural ways. Furthermore, if multiple users share a common file server, as is the case on many company networks, each user can have a private root directory for his or her own hierarchy. This approach is shown in Fig. 4-7. Here, the directories *A*, *B*, and *C* contained in the root directory each belong to a different user, two of whom have created subdirectories for projects they are working on.

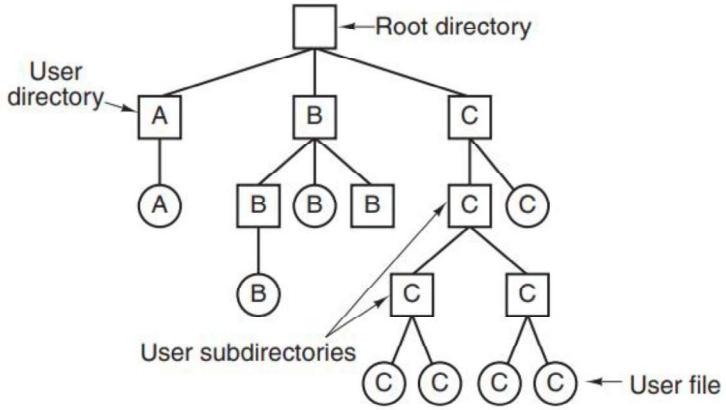

Figure 4-7. A hierarchical directory system.

The ability for users to create an arbitrary number of subdirectories provides a powerful structuring tool for users to organize their work. For this reason, nearly all modern file systems are organized in this manner.

Path Names

When the file system is organized as a directory tree, some way is needed for specifying file names. Two different methods are commonly used. In the first method, each file **is given an absolute path name** consisting of the path from the root directory to the file. As an example, the path /usr/ast/mailbox means that the root directory contains a subdirectory usr, which in turn contains a subdirectory ast, which contains the file mailbox. Absolute path names always start at the root directory and are unique. In UNIX the components of the path are separated by /. In Windows the separator is \ . In MULTICS it was >. Thus, the same path name would be written as follows in these three systems:

Windows \usr\ast\mailbox

UNIX /usr/ast/mailbox

MULTICS >usr>ast>mailbox

No matter which character is used, if the first character of the path name is the separator, then the path is absolute.

The other kind of name is the relative path name. This is used in conjunction with the concept of the working directory (also called the current directory). A user can designate one directory as the current working directory, in which case all path names not beginning at the root directory are taken relative to the working directory. For example, if the current working directory is /usr/ast, then the file whose absolute path is /usr/ast/mailbox can be referenced simply as mailbox. In other words, the UNIX command .

cp /usr/ast/mailbox /usr/ast/mailbox.bak

and the command

cp mailbox mailbox.bak

Most operating systems that support a hierarchical directory system have two special entries in every directory, ‘‘.’’ and ‘‘..’’, generally pronounced ‘‘dot’’ and ‘‘dotdot.’’ Dot refers to the current directory;

dotdot refers to its parent (except in the root directory, where it refers to itself). To see how these are used, consider the UNIX file tree of Fig. 4-8. A certain process has /usr/ast as its working directory. It can use .. to go higher up the tree. For example, it can copy the file /usr/lib/dictionary to its own directory using the command .

cp ../lib/dictionary .

The first path instructs the system to go upward (to the usr directory), then to go down to the directory lib to find the file dictionary.

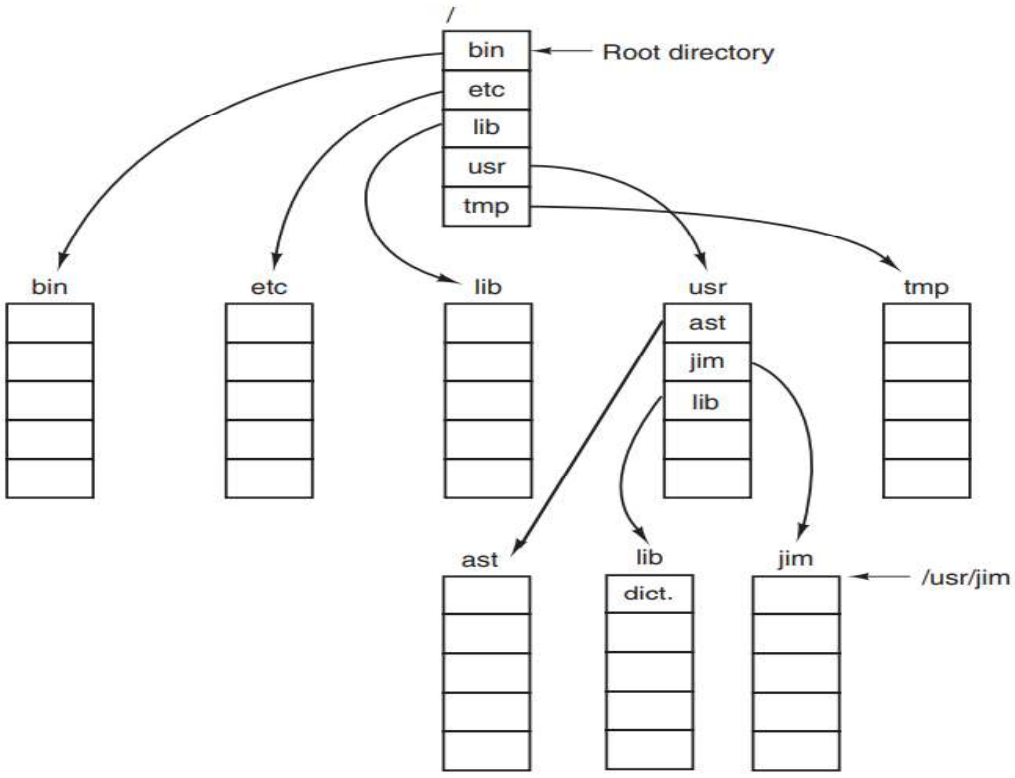

Figure 4-8. A UNIX directory tree.

The second argument (dot) names the current directory. When the cp command gets a directory name (including dot) as its last argument, it copies all the files to that directory. Of course, a more normal way to do the copy would be to use the full absolute path name of the source file:

cp /usr/lib/dictionary .

Here the use of dot saves the user the trouble of typing dictionary a second time. Nevertheless, typing

cp /usr/lib/dictionary dictionar y

also works fine, as does

cp /usr/lib/dictionary /usr/ast/dictionar y

All of these do exactly the same thing.

Directory Operations

The allowed system calls for managing directories exhibit more variation from system to system than system calls for files. To giv e an impression of what they are and how they work, we will give a sample (taken from UNIX).

1. **Create.** A directory is created. It is empty except for dot and dotdot, which are put there automatically by the system (or in a few cases, by the mkdir program).

2. **Delete.** A directory is deleted. Only an empty directory can be deleted. A directory containing only dot and dotdot is considered empty as these cannot be deleted.

3. **Opendir**. Directories can be read. For example, to list all the files in a directory, a listing program opens the directory to read out the names of all the files it contains. Before a directory can be read, it must be opened, analogous to opening and reading a file.

4. **Closedir**. When a directory has been read, it should be closed to free up internal table space.

5. **Readdir**. This call returns the next entry in an open directory. Formerly, it was possible to read directories using the usual read system call, but that approach has the disadvantage of forcing the programmer to know and deal with the internal structure of directories. In contrast, readdir always returns one entry in a standard format, no matter which of the possible directory structures is being used.

6. **Rename**. In many respects, directories are just like files and can be renamed the same way files can be.

7. Link. Linking is a technique that allows a file to appear in more than one directory. This system call specifies an existing file and a path SEC. 4.2 DIRECTORIES 281 name, and creates a link from the existing file to the name specified by the path. In this way, the same file may appear in multiple directories. A link of this kind, which increments the counter in the file's i-node (to keep track of the number of directory entries containing the file), is sometimes called a **hard link.**

8. Unlink. A directory entry is removed. If the file being unlinked is only present in one directory (the normal case), it is removed from the file system. If it is present in multiple directories, only the path name specified is removed. The others remain. In UNIX, the system call for deleting files (discussed earlier) is, in fact, unlink.

The above list gives the most important calls, but there are a few others as well, for example, for managing the protection information associated with a directory.

A variant on the idea of linking files is the symbolic link. Instead, of having two names point to the same internal data structure representing a file, a name can be created that points to a tiny file naming another file. When the first file is used, for example, opened, the file system follows the path and finds the name at the end. Then it starts the lookup process all over using the new name. Symbolic links have the advantage that they can cross disk boundaries and even name files on remote computers. Their implementation is somewhat less efficient than hard links though.

3)FILE-SYSTEM IMPLEMENTATION :

Now it is time to turn from the user's view of the file system to the implementor's view. Users are concerned with how files are named, what operations are allowed on them, what the directory tree looks like, and similar interface issues. Implementors are interested in how files and directories are stored, how disk space is managed, and how to make everything work efficiently and reliably. In the following sections we will examine a number of these areas to see what the issues and trade-offs are.

File-System Layout

File systems are stored on disks. Most disks can be divided up into one or more partitions, with independent file systems on each partition. Sector 0 of the disk is called the **MBR (Master Boot Record**) and is used to boot the computer. The end of the MBR contains the partition table. This table gives the starting and ending addresses of each partition. One of the partitions in the table is marked as active. When the computer is booted, the BIOS reads in and executes the MBR. The first thing the MBR program does is

locate the active partition, read in its first block, which is called the **boot block**, and execute it. The program in the boot block loads the operating system contained in that partition. For uniformity, every partition starts with a boot block, even if it does not contain a bootable operating system. Besides, it might contain one in the future.

Other than starting with a boot block, the layout of a disk partition varies a lot from file system to file system. Often the file system will contain some of the items shown in Fig. 4-9. The first one is the superblock. It contains all the key parameters about the file system and is read into memory when the computer is booted or the file system is first touched. Typical information in the superblock includes a magic number to identify the file-system type, the number of blocks in the file system, and other key administrative information.

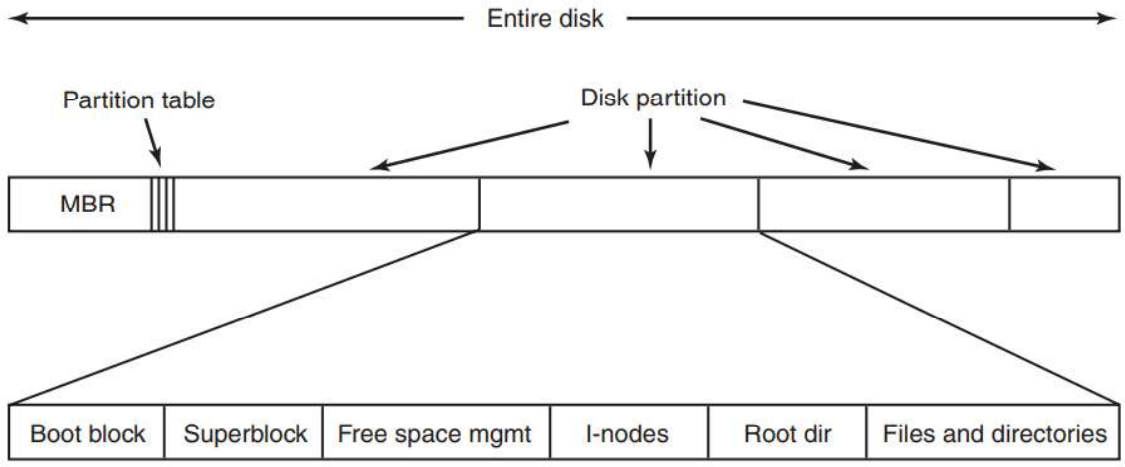

Figure 4-9. A possible file-system layout.

Next might come information about free blocks in the file system, for example in the form of a bitmap or a list of pointers. This might be followed by the i-nodes, an array of data structures, one per file, telling all about the file. After that might come the root directory, which contains the top of the file-system tree. Finally, the remainder of the disk contains all the other directories and files.

Implementing Files

Probably the most important issue in implementing file storage is keeping track of which disk blocks go with which file. Various methods are used in different operating systems. In this section, we will examine a few of them.

Contiguous Allocation

The simplest allocation scheme is to store each file as a contiguous run of disk blocks. Thus on a disk with 1-KB blocks, a 50-KB file would be allocated 50 consecutive blocks. With 2-KB blocks, it would be allocated 25 consecutive blocks. We see an example of contiguous storage allocation in Fig. 4-10(a). Here the first 40 disk blocks are shown, starting with block 0 on the left. Initially, the disk was empty. Then a file A, of length four blocks, was written to disk starting at the beginning (block 0). After that a six-block file, B, was written starting right after the end of file A.

Note that each file begins at the start of a new block, so that if file A was really 3½ blocks, some space is wasted at the end of the last block. In the figure, a total of seven files are shown, each one starting at the block following the end of the previous one. Shading is used just to make it easier to tell the files apart. It has no actual significance in terms of storage..

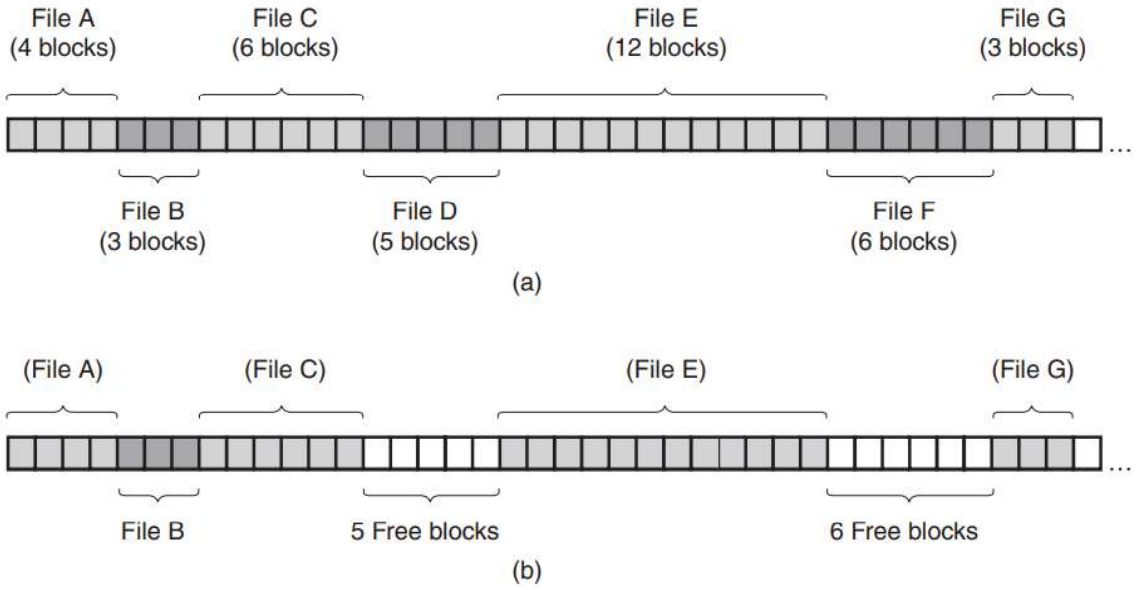

Figure 4-10. (a) Contiguous allocation of disk space for seven files. (b) The state of the disk after files *D* and *F* have been removed.

Contiguous disk-space allocation has two significant advantages. First, it is simple to implement because keeping track of where a file's blocks are is reduced to remembering two numbers: the disk address of the first block and the number of blocks in the file. Given the number of the first block, the number of any other block can be found by a simple addition.

Imagine the consequences of such a design. The user starts a word processor in order to create a document. The first thing the program asks is how many bytes the final document will be. The question must be answered or the program will not continue. If the number given ultimately proves too small, the program has to terminate prematurely because the disk hole is full and there is no place to put the rest of the file. If the user tries to avoid this problem by giving an unrealistically large number as the final size, say, 1 GB, the editor may be unable to find such a large hole and announce that the file cannot be created. Of course, the user would be free to start the program again and say 500 MB this time, and so on until a suitable hole was located. Still, this scheme is not likely to lead to happy users.

However, there is one situation in which contiguous allocation is feasible and, in fact, still used: on CD-ROMs. Here all the file sizes are known in advance and will never change during subsequent use of the CD-ROM file system.

The situation with DVDs is a bit more complicated. In principle, a 90-min movie could be encoded as a single file of length about 4.5 GB, but the file system used, UDF (Universal Disk Format), uses a 30-bit number to represent file length, which limits files to 1 GB. As a consequence, DVD movies are generally stored as three or four 1-GB files, each of which is contiguous. These physical pieces of the single logical file (the movie) are called **extents.**

As we mentioned in Chap. 1, history often repeats itself in computer science as new generations of technology occur. Contiguous allocation was actually used on magnetic-disk file systems years ago due to its simplicity and high performance (user friendliness did not count for much then). Then the idea was dropped due to the nuisance of having to specify final file size at file-creation time. But with the advent of CD-ROMs, DVDs, Blu-rays, and other write-once optical media, suddenly contiguous files were a good idea again. It is thus important to study old systems and ideas that were conceptually clean and simple because they may be applicable to future systems in surprising ways.

Linked-List Allocation

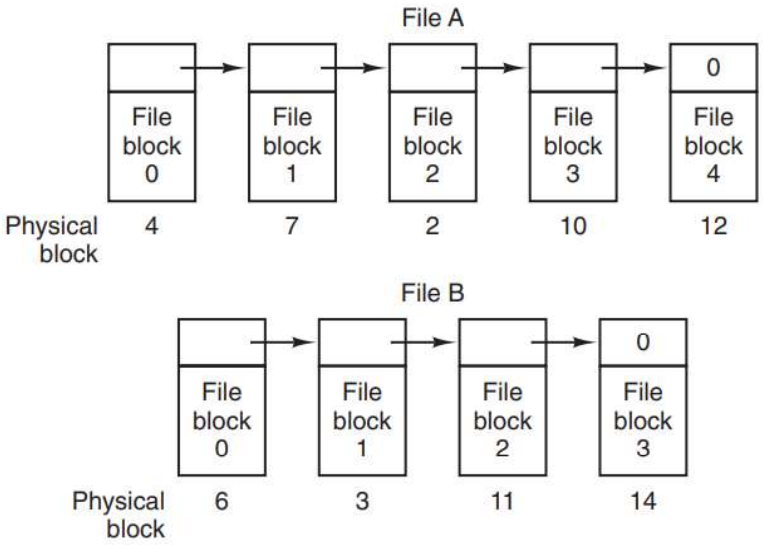

Figure 4-11. Storing a file as a linked list of disk blocks.

The second method for storing files is to keep each one as a linked list of disk blocks, as shown in Fig. 4-11. The first word of each block is used as a pointer to the next one. The rest of the block is for data.

Unlike contiguous allocation, every disk block can be used in this method. No space is lost to disk fragmentation (except for internal fragmentation in the last block). Also, it is sufficient for the directory entry to merely store the disk address of the first block. The rest can be found starting there. On the other hand, although reading a file sequentially is straightforward, random access is extremely slow. To get to block n, the operating system has to start at the beginning and read the n − 1 blocks prior to it, one at a time. Clearly, doing so many reads will be painfully slow.

Also, the amount of data storage in a block is no longer a power of two because the pointer takes up a few bytes. While not fatal, having a peculiar size is less efficient because many programs read and write in blocks whose size is a power of two. With the first few bytes of each block occupied by a pointer to the next block, reads of the full block size require acquiring and concatenating information from two disk blocks, which generates extra overhead due to the copying.

Linked-List Allocation Using a Table in Memory :

Both disadvantages of the linked-list allocation can be eliminated by taking the pointer word from each disk block and putting it in a table in memory. Figure 4-12 shows what the table looks like for the example of Fig. 4-11. In both figures, we have two files. File A uses disk blocks 4, 7, 2, 10, and 12, in that order, and file B uses disk blocks 6, 3, 11, and 14, in that order. Using the table of Fig. 4-12, we can start with block 4 and follow the chain all the way to the end. The same can be done starting with block 6. Both chains are terminated with a special marker (e.g., −1) that is not a valid block number. Such a table in main memory is called a FAT (File Allocation Table).

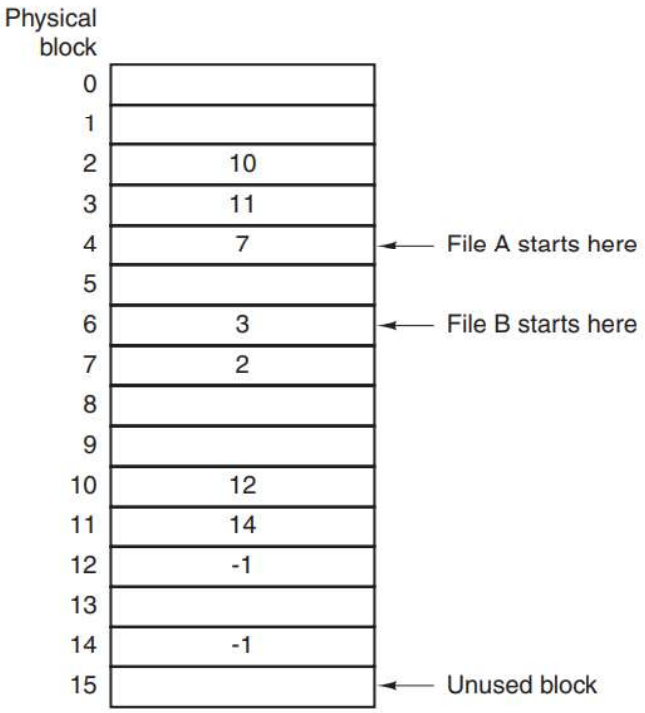

Figure 4-12. Linked-list allocation using a file-allocation table in main memory.

Using this organization, the entire block is available for data. Furthermore, random access is much easier. Although the chain must still be followed to find a given offset within the file, the chain is entirely in memory, so it can be followed without making any disk references. Like the previous method, it is sufficient for the directory entry to keep a single integer (the starting block number) and still be able to locate all the blocks, no matter how large the file is.

The primary disadvantage of this method is that the entire table must be in memory all the time to make it work. With a 1-TB disk and a 1-KB block size, the table needs 1 billion entries, one for each of the 1 billion disk blocks. Each entry has to be a minimum of 3 bytes. For speed in lookup, they should be 4 bytes. Thus the table will take up 3 GB or 2.4 GB of main memory all the time, depending on whether the system is optimized for space or time. Not wildly practical. Clearly the FAT idea does not scale well to large disks. It was the original MS-DOS file system and is still fully supported by all versions of Windows though.

I-nodes

Our last method for keeping track of which blocks belong to which file is to associate with each file a data structure called an **i-node (index-node),** which lists the attributes and disk addresses of the file's blocks. A simple example is depicted in Fig. 4-13. Given the i-node, it is then possible to find all the blocks of the file.

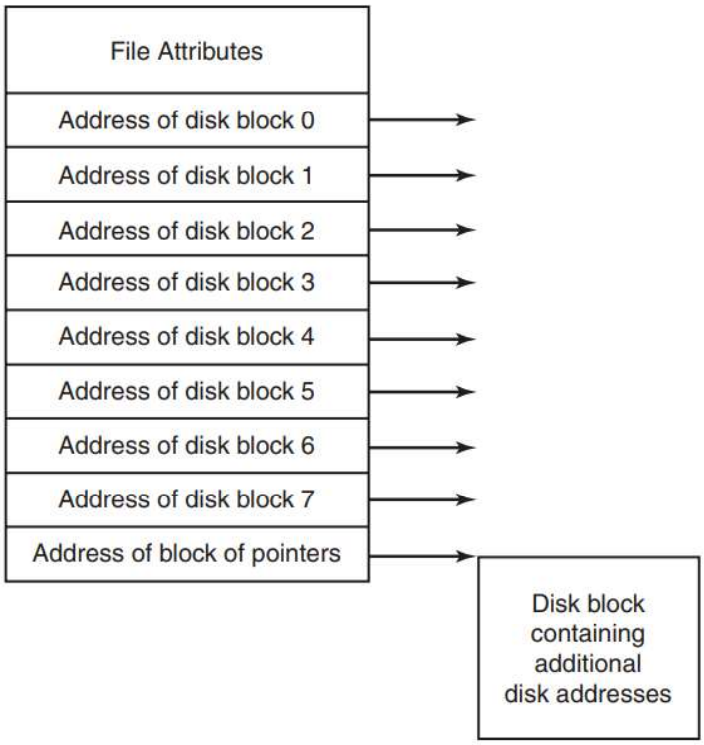

Figure 4-13. An example i-node.

This array is usually far smaller than the space occupied by the file table described in the previous section. The reason is simple. The table for holding the linked list of all disk blocks is proportional in size to the disk itself. If the disk has n blocks, the table needs n entries. As disks grow larger, this table grows linearly with them. In contrast, the i-node scheme requires an array in memory whose size is proportional to the maximum number of files that may be open at once. It does not matter if the disk is 100 GB, 1000 GB, or 10,000 GB.

One problem with i-nodes is that if each one has room for a fixed number of disk addresses, what happens when a file grows beyond this limit? One solution is to reserve the last disk address not for a data block, but instead for the address of a block containing more disk-block addresses, as shown in Fig. 4-13. Even more advanced would be two or more such blocks containing disk addresses or even disk blocks pointing to other disk blocks full of addresses. We will come back to **inodes** when studying UNIX in Chap. 10. Similarly, the Windows NTFS file system uses a similar idea, only with bigger i-nodes that can also contain small files

Implementing Directories:

Before a file can be read, it must be opened. When a file is opened, the operating system uses the path name supplied by the user to locate the directory entry on the disk. The directory entry provides the information needed to find the disk blocks. Depending on the system, this information may be the disk address of the entire file (with contiguous allocation), the number of the first block (both linked-list schemes), or the number of the i-node. In all cases, the main function of the directory system is to map the ASCII name of the file onto the information needed to locate the data. A closely related issue is where the attributes should be stored. Every file system maintains various file attributes, such as each file's owner and creation time, and they must be stored somewhere. One obvious possibility is to store them directly in the directory entry. Some systems do precisely that. This option is shown in Fig. 4-14(a). In this simple design, a directory consists of a list of fixed-size entries, one per file, containing a (fixed-length) file name, a structure of the file attributes, and one or more disk addresses (up to some maximum) telling where the disk blocks are.

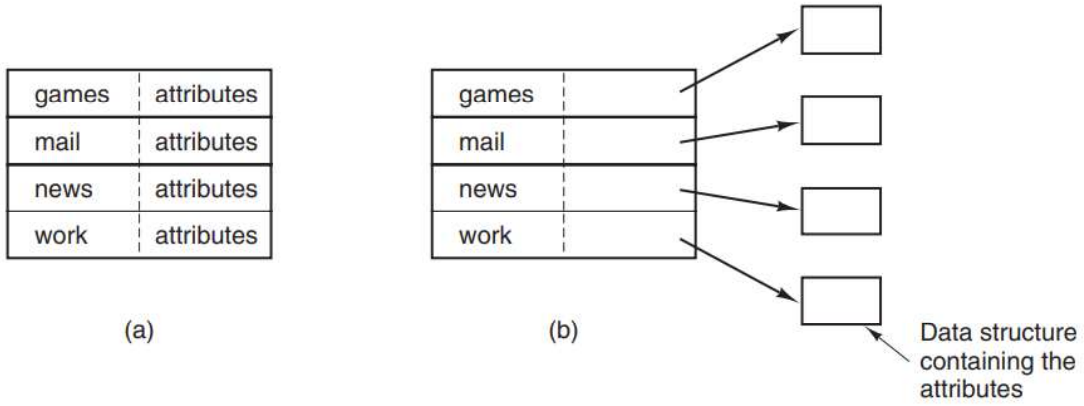

Figure 4-14. (a) A simple directory containing fixed-size entries with the disk addresses and attributes in the directory entry. (b) A directory in which each entry just refers to an i-node.

So far we have made the assumption that files have short, fixed-length names. In MS-DOS files have a 1–8 character base name and an optional extension of 1–3 characters. In UNIX Version 7, file names were 1–14 characters, including any extensions. However, nearly all modern operating systems support longer, variable-length file names. How can these be implemented?.

One alternative is to give up the idea that all directory entries are the same size. With this method, each directory entry contains a fixed portion, typically starting with the length of the entry, and then followed by data with a fixed format, usually including the owner, creation time, protection information, and other attributes. This fixed-length header is followed by the actual file name, however long it may be, as shown in Fig. 4-15(a) in big-endian format (e.g., SPARC). In this example we have three files, project-budget, personnel, and foo. Each file name is terminated by a special character (usually 0), which is represented in the figure by a box with a cross in it. To allow each directory entry to begin on a word boundary, each file name is filled out to an integral number of words, shown by shaded boxes in the figure.

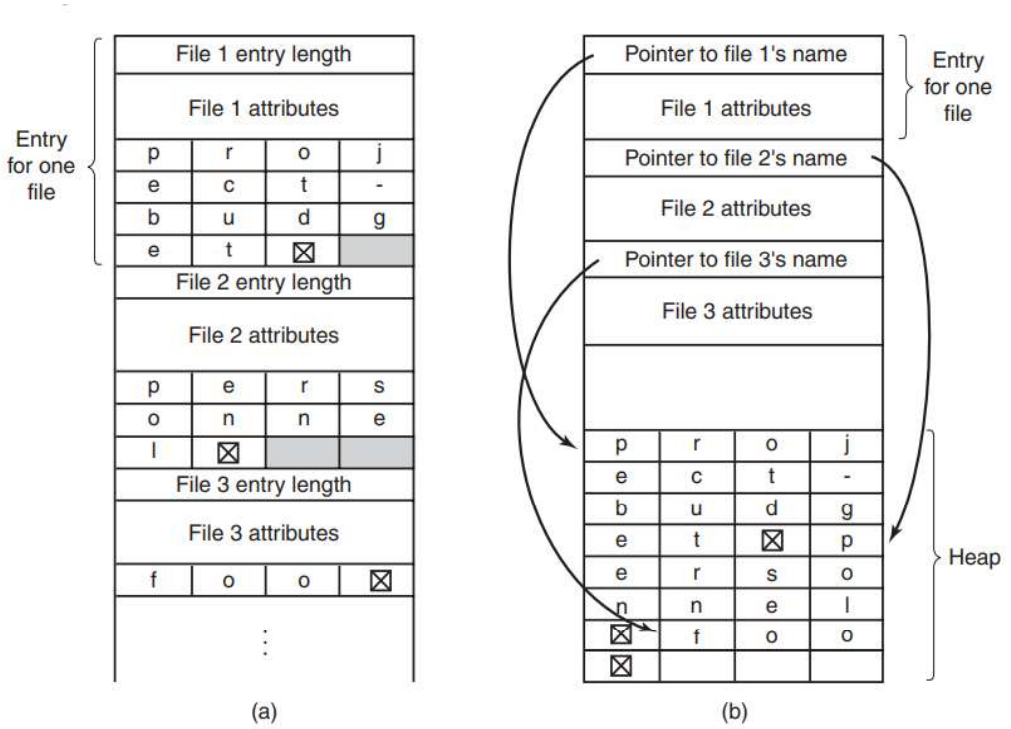

Figure 4-15. Two ways of handling long file names in a directory. (a) In-line. (b) In a heap.

A disadvantage of this method is that when a file is removed, a variable-sized gap is introduced into the directory into which the next file to be entered may not fit. This problem is essentially the same one we saw with contiguous disk files, only now compacting the directory is feasible because it is entirely in memory. Another problem is that a single directory entry may span multiple pages, so a page fault may occur while reading a file name.

Looking up a file follows the same procedure. The file name is hashed to select a hash-table entry. All the entries on the chain headed at that slot are checked to see if the file name is present. If the name is not on the chain, the file is not present in the directory.
Using a hash table has the advantage of much faster lookup, but the disadvantage of more complex administration. It is only really a serious candidate in systems where it is expected that directories will routinely contain hundreds or thousands of files.
A different way to speed up searching large directories is to cache the results of searches. Before starting a search, a check is first made to see if the file name is in the cache. If so, it can be located immediately. Of course, caching only works if a relatively small number of files comprise the majority of the lookups..

Shared Files

When several users are working together on a project, they often need to share files. As a result, it is often convenient for a shared file to appear simultaneously in different directories belonging to different users. Figure 4-16 shows the file system of Fig. 4-7 again, only with one of C's files now present in one of B's directories as well. The connection between B's directory and the shared file is called a link. The file system itself is now a Directed Acyclic Graph, or DAG, rather than a tree. Having the file system be a DAG complicates maintenance, but such is life.

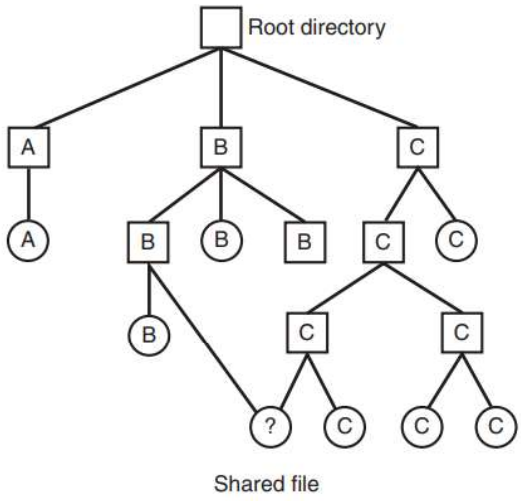

Figure 4-16. File system containing a shared file.

Sharing files is convenient, but it also introduces some problems. To start with, if directories really do contain disk addresses, then a copy of the disk addresses will have to be made in B's directory when the file is linked. If either B or C subsequently appends to the file, the new blocks will be listed only in the directory of the user doing the append. The changes will not be visible to the other user, thus defeating the purpose of sharing.
This problem can be solved in two ways. In the first solution, disk blocks are not listed in directories, but in a little data structure associated with the file itself. The directories would then point just to the little data structure. This is the approach used in UNIX (where the little data structure is the i-node).
In the second solution, B links to one of C's files by having the system create a new file, of type LINK, and entering that file in B's directory. The new file contains just the path name of the file to which it is linked. When B reads from the linked file, the operating system sees that the file being read from is of type LINK,

looks up the name of the file, and reads that file. This approach is called symbolic linking, to contrast it with traditional (hard) linking.

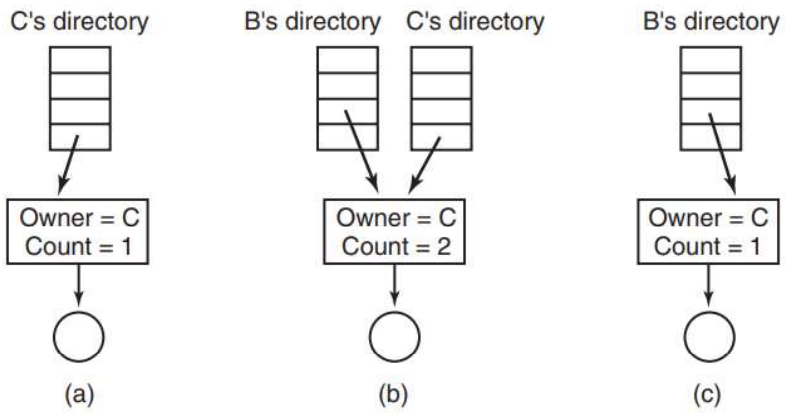

Figure 4-17. (a) Situation prior to linking. (b) After the link is created. (c) After the original owner removes the file.

an invalid i-node. If the i-node is later reassigned to another file, B's link will point to the wrong file. The system can see from the count in the i-node that the file is still in use, but there is no easy way for it to find all the directory entries for the file, in order to erase them. Pointers to the directories cannot be stored in the inode because there can be an unlimited number of directories.

The only thing to do is remove C's directory entry, but leave the i-node intact, with count set to 1, as shown in Fig. 4-17(c). We now hav e a situation in which B is the only user having a directory entry for a file owned by C. If the system does accounting or has quotas, C will continue to be billed for the file until B decides to remove it, if ever, at which time the count goes to 0 and the file is deleted.

With symbolic links this problem does not arise because only the true owner has a pointer to the i-node. Users who have linked to the file just have path names, not i-node pointers. When the owner removes the file, it is destroyed. Subsequent attempts to use the file via a symbolic link will fail when the system is unable to locate the file. Removing a symbolic link does not affect the file at all.

The problem with symbolic links is the extra overhead required. The file containing the path must be read, then the path must be parsed and followed, component by component, until the i-node is reached. All of this activity may require a considerable number of extra disk accesses. Furthermore, an extra i-node is needed for each symbolic link, as is an extra disk block to store the path, although if the path name is short, the system could store it in the i-node itself, as a kind of optimization. Symbolic links have the advantage that they can be used to link to files on machines anywhere in the world, by simply providing the network address of the machine where the file resides in addition to its path on that machine. There is also another problem introduced by links, symbolic or otherwise. When links are allowed, files can have two or more paths. Programs that start at a given directory and find all the files in that directory and its subdirectories will locate a linked file multiple times. For example, a program that dumps all the files in a directory and its subdirectories onto a tape may make multiple copies of a linked file. Furthermore, if the tape is then read into another machine, unless the dump program is clever, the linked file will be copied twice onto the disk, instead of being linked.

Log-Structured File Systems :

Changes in technology are putting pressure on current file systems. In particular, CPUs keep getting faster, disks are becoming much bigger and cheaper (but not much faster), and memories are growing

exponentially in size. The one parameter that is not improving by leaps and bounds is disk seek time (except for solid-state disks, which have no seek time).

The combination of these factors means that a performance bottleneck is arising in many file systems. Research done at Berkeley attempted to alleviate this problem by designing a completely new kind of file system, LFS (the Log-structured File System). In this section we will briefly describe how LFS works. For a more complete treatment, see the original paper on LFS (Rosenblum and Ousterhout, 1991).+

From this reasoning, the LFS designers decided to reimplement the UNIX file system in such a way as to achieve the full bandwidth of the disk, even in the face of a workload consisting in large part of small random writes. The basic idea is to structure the entire disk as a great big log. Periodically, and when there is a special need for it, all the pending writes being buffered in memory are collected into a single segment and written to the disk as a single contiguous segment at the end of the log.

To deal with this problem, LFS has a cleaner thread that spends its time scanning the log circularly to compact it. It starts out by reading the summary of the first segment in the log to see which i-nodes and files are there. It then checks the current i-node map to see if the i-nodes are still current and file blocks are still in use. If not, that information is discarded. The i-nodes and blocks that are still in use go into memory to be written out in the next segment. The original segment is then marked as free, so that the log can use it for new data. In this manner, the cleaner moves along the log, removing old segments from the back and putting any live data into memory for rewriting in the next segment. Consequently, the disk is a big circular buffer, with the writer thread adding new segments to the front and the cleaner thread removing old ones from the back.

The bookkeeping here is nontrivial, since when a file block is written back to a new segment, the i-node of the file (somewhere in the log) must be located, updated, and put into memory to be written out in the next segment. The i-node map must then be updated to point to the new copy. Nev ertheless, it is possible to do the administration, and the performance results show that all this complexity is worthwhile. Measurements given in the papers cited above show that LFS outperforms UNIX by an order of magnitude on small writes, while having a performance that is as good as or better than UNIX for reads and large writes.

Journaling File Systems

While log-structured file systems are an interesting idea, they are not widely used, in part due to their being highly incompatible with existing file systems. Nevertheless, one of the ideas inherent in them, robustness in the face of failure, can be easily applied to more conventional file systems. The basic idea here is to keep a log of what the file system is going to do before it does it, so that if the system crashes before it can do its planned work, upon rebooting the system can look in the log to see what was going on at the time of the crash and finish the job. Such file systems, called journaling file systems, are actually in use. Microsoft's NTFS file system and the Linux ext3 and ReiserFS file systems all use journaling. OS X offers journaling file systems as an option. Below we will give a brief introduction to this topic.

To see the nature of the problem, consider a simple garden-variety operation that happens all the time: removing a file. This operation (in UNIX) requires three steps:

1. Remove the file from its directory.
2. Release the i-node to the pool of free i-nodes.
3. Return all the disk blocks to the pool of free disk blocks.

In Windows analogous steps are required. In the absence of system crashes, the order in which these steps are taken does not matter; in the presence of crashes, it does. Suppose that the first step is completed and then the system crashes. The inode and file blocks will not be accessible from any file, but will also not be available for reassignment; they are just off in limbo somewhere, decreasing the available resources. If the crash occurs after the second step, only the blocks are lost.

For added reliability, a file system can introduce the **database concept of an atomic transaction**. When this concept is used, a group of actions can be bracketed by the begin transaction and end transaction operations. The file system then knows it must complete either all the bracketed operations or none of them, but not any other combinations.

Virtual File Systems

Many different file systems are in use—often on the same computer—even for the same operating system. A Windows system may have a main NTFS file system, but also a legacy FAT -32 or FAT -16 drive or partition that contains old, but still needed, data, and from time to time a flash drive, an old CD-ROM or a DVD (each with its own unique file system) may be required as well. Windows handles these disparate file systems by identifying each one with a different drive letter, as in C:, D:, etc. When a process opens a file, the drive letter is explicitly or implicitly present so Windows knows which file system to pass the request to. There is no attempt to integrate heterogeneous file systems into a unified whole.

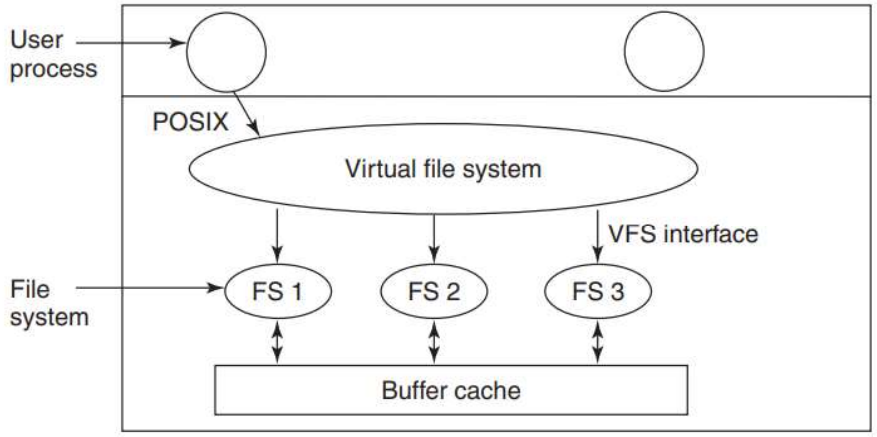

Figure 4-18. Position of the virtual file system.

most UNIX systems have used the concept of a VFS (virtual file system) to try to integrate multiple file systems into an orderly structure. The key idea is to abstract out that part of the file system that is common to all file systems and put that code in a separate layer that calls the underlying concrete file systems to actually manage the data. The overall structure is illustrated in Fig. 4-18. The discussion below is not specific to Linux or FreeBSD or any other version of UNIX, but giv es the general flavor of how virtual file systems work in UNIX systems.

After a file system has been mounted, it can be used. For example, if a file system has been mounted on /usr and a process makes the call

open("/usr/include/unistd.h", O RDONLY)

while parsing the path, the VFS sees that a new file system has been **mounted on /usr and locates its superblock by searching the list of superblocks** of mounted file systems. Having done this, it can find the root directory of the mounted file system and look up the path include/unistd.h there. The VFS then creates a v-node and makes a call to the concrete file system to return all the information in the file's inode. This information is copied into the v-node (in RAM), along with other information, most importantly the pointer to the table of functions to call for operations on v-nodes, such as read, wr ite, close, and so on.

are shown in Fig. 4-19. Starting with the caller's process number and the file descriptor, successively the v-node, read function pointer, and access function within the concrete file system are located.

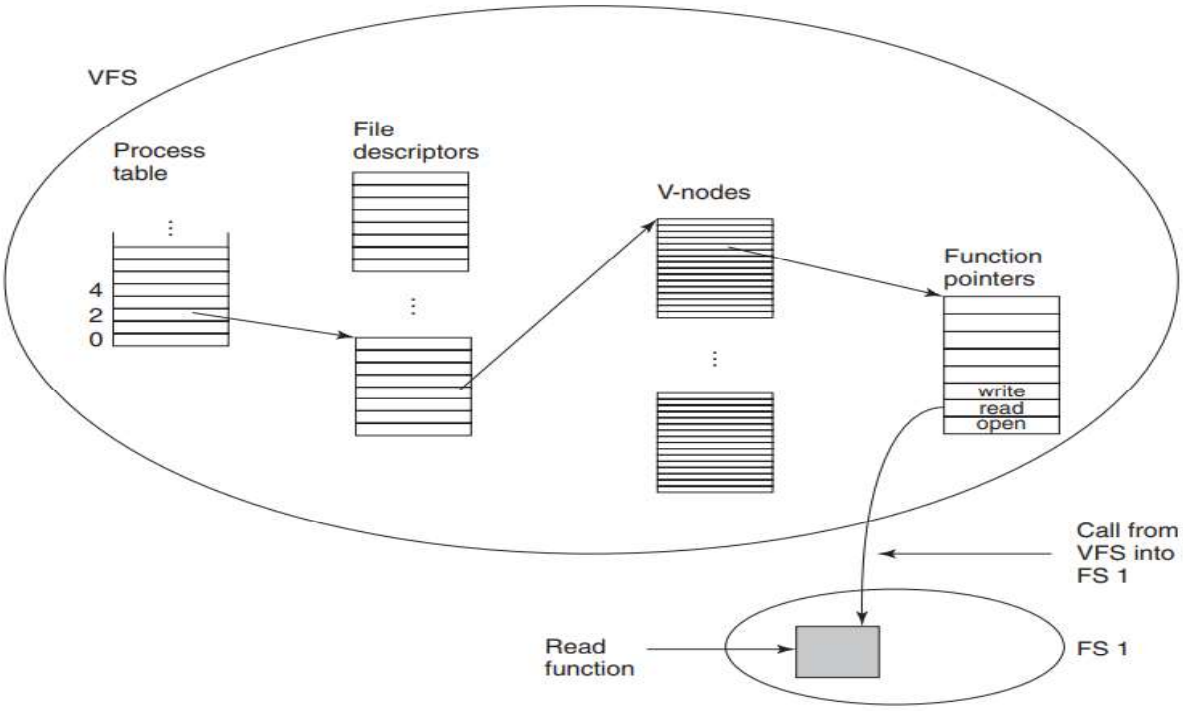

Figure 4-19. A simplified view of the data structures and code used by the VFS and concrete file system to do a read.

4)FILE-SYSTEM MANAGEMENT AND OPTIMIZATION

Making the file system work is one thing; making it work efficiently and robustly in real life is something quite different. In the following sections we will look at some of the issues involved in managing disks.

Disk-Space:

Management Files are normally stored on disk, so management of disk space is a major concern to file-system designers. Two general strategies are possible for storing an n byte file: n consecutive bytes of disk space are allocated, or the file is split up into a number of (not necessarily) contiguous blocks. The same trade-off is present in memory-management systems between pure segmentation and paging.

As we have seen, storing a file as a contiguous sequence of bytes has the obvious problem that if a file grows, it may have to be moved on the disk. The same problem holds for segments in memory, except that moving a segment in memory is a relatively fast operation compared to moving a file from one disk position to another. For this reason, nearly all file systems chop files up into fixed-size blocks that need not be adjacent.

Block Size

Once it has been decided to store files in fixed-size blocks, the question arises how big the block should be. Given the way disks are organized, the sector, the track, and the cylinder are obvious candidates for the unit of allocation (although these are all device dependent, which is a minus). In a paging system, the page size is also a major contender. Having a large block size means that every file, even a 1-byte file, ties up an entire cylinder. It also means that small files waste a large amount of disk space. On the other hand, a small block size means that most files will span multiple blocks and thus need multiple seeks and rotational delays to read them, reducing performance. Thus if the allocation unit is too large, we waste space; if it is too small, we waste time.

The dashed curve of Fig. 4-21 shows the data rate for such a disk as a function of block size. To compute the space efficiency, we need to make an assumption about the mean file size. For simplicity, let us assume that all files are 4 KB. Although this number is slightly larger than the data measured at the VU, students probably have more small files than would be present in a corporate data center, so it might be a better guess on the whole. The solid curve of Fig. 4-21 shows the space efficiency as a function of block size.

The two curves can be understood as follows. The access time for a block is completely dominated by the seek time and rotational delay, so giv en that it is going to cost 9 msec to access a block, the more data that are fetched, the better.

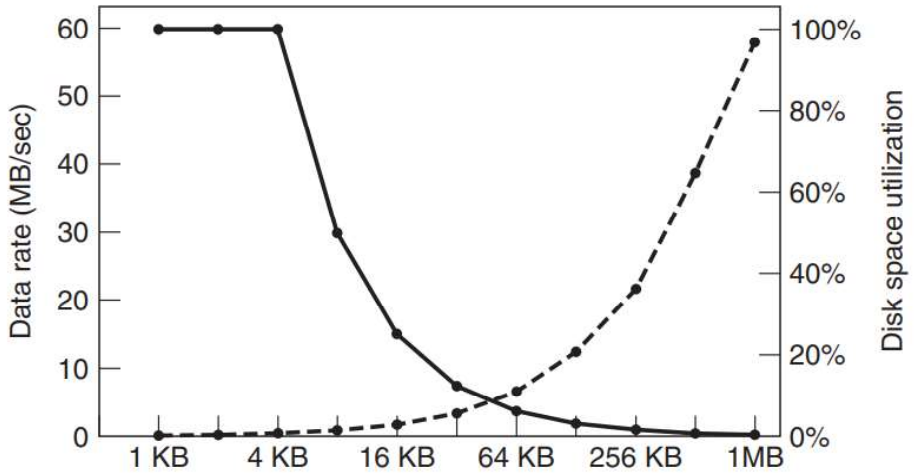

Figure 4-21. The dashed curve (left-hand scale) gives the data rate of a disk. The solid curve (right-hand scale) gives the disk-space efficiency. All files are 4 KB.

Hence the data rate goes up almost linearly with block size (until the transfers take so long that the transfer time begins to matter).

Now consider space efficiency. With 4-KB files and 1-KB, 2-KB, or 4-KB blocks, files use 4, 2, and 1 block, respectively, with no wastage. With an 8-KB block and 4-KB files, the space efficiency drops to 50%, and with a 16-KB block it is down to 25%. In reality, few files are an exact multiple of the disk block size, so some space is always wasted in the last block of a file.

What the curves show, however, is that performance and space utilization are inherently in conflict. Small blocks are bad for performance but good for diskspace utilization. For these data, no reasonable compromise is available. The size closest to where the two curves cross is 64 KB, but the data rate is only 6.6 MB/sec and the space efficiency is about 7%, neither of which is very good. Historically, file systems have chosen sizes in the 1-KB to 4-KB range, but with disks now exceeding 1 TB, it might be better to increase the block size to 64 KB and accept the wasted disk space. Disk space is hardly in short supply any more.

In an experiment to see if Windows NT file usage was appreciably different from UNIX file usage, Vogels made measurements on files at Cornell University (Vogels, 1999). He observed that NT file usage is more complicated than on UNIX. He wrote:

When we type a few characters in the Notepad text editor, saving this to a file will trigger 26 system calls, including 3 failed open attempts, 1 file overwrite and 4 additional open and close sequences.

Keeping Track of Free Blocks

Once a block size has been chosen, the next issue is how to keep track of free blocks. Two methods are widely used, as shown in Fig. 4-22. The first one consists of using a linked list of disk blocks, with each block holding as many free disk block numbers as will fit. With a 1-KB block and a 32-bit disk block number, each block on the free list holds the numbers of 255 free blocks. (One slot is required for the pointer to the next block.) Consider a 1-TB disk, which has about 1 billion disk blocks. To store all these addresses at 255 per block requires about 4 million blocks. Generally, free blocks are used to hold the free list, so the storage is essentially free.

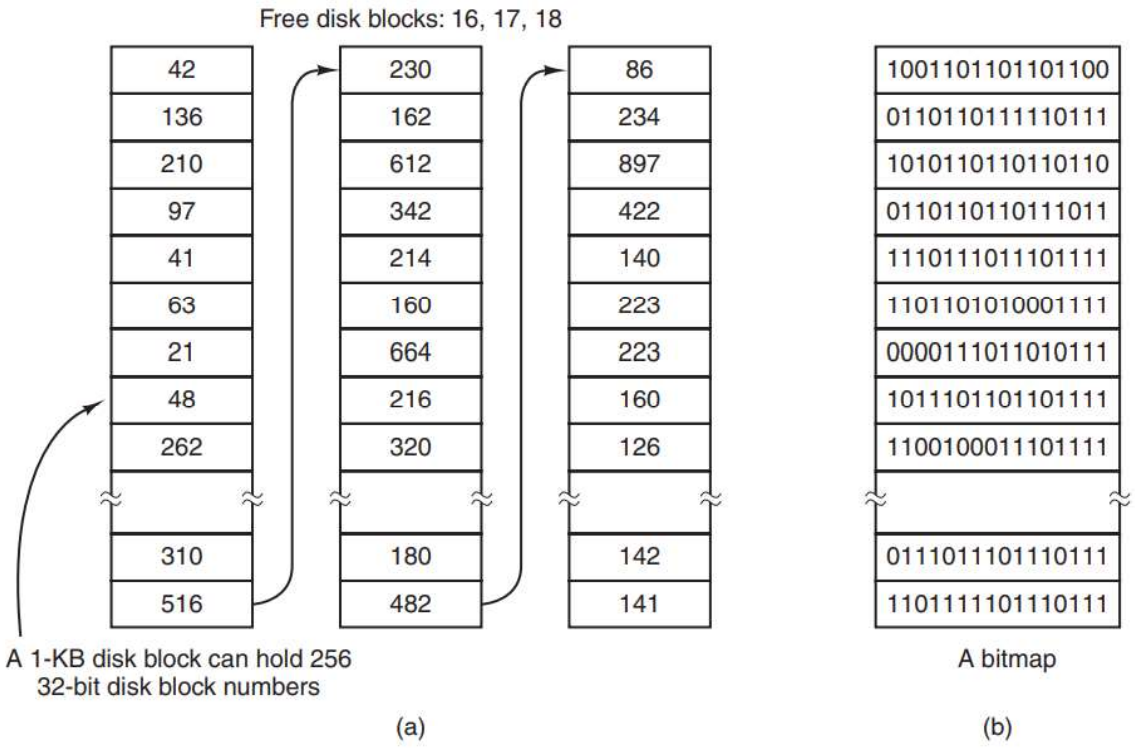

Figure 4-22. (a) Storing the free list on a linked list. (b) A bitmap.

The other free-space management technique is the bitmap. A disk with n blocks requires a bitmap with n bits. Free blocks are represented by 1s in the map, allocated blocks by 0s (or vice versa). For our example 1-TB disk, we need 1 billion bits for the map, which requires around 130,000 1-KB blocks to store. It is not surprising that the bitmap requires less space, since it uses 1 bit per block, vs. 32 bits in the linked-list model. Only if the disk is nearly full (i.e., has few free blocks) will the linked-list scheme require fewer blocks than the bitmap.

If free blocks tend to come in long runs of consecutive blocks, the free-list system can be modified to keep track of runs of blocks rather than single blocks. An 8-, 16-, or 32-bit count could be associated with each block giving the number of consecutive free blocks. In the best case, a basically empty disk could be represented by two numbers: the address of the first free block followed by the count of free blocks. On the other hand, if the disk becomes severely fragmented, keeping track of runs is less efficient than keeping track of individual blocks because not only must the address be stored, but also the count.

An alternative approach that avoids most of this disk I/O is to split the full block of pointers. Thus instead of going from Fig. 4-23(a) to Fig. 4-23(b), we go from Fig. 4-23(a) to Fig. 4-23(c) when three blocks are freed. Now the system can handle a series of temporary files without doing any disk I/O. If the block in memory fills up, it is written to the disk, and the half-full block from the disk is read in. The idea here is to keep most of the pointer blocks on disk full (to minimize disk usage), but keep the one in memory about half full, so it can handle both file creation and file removal without disk I/O on the free list.

With a bitmap, it is also possible to keep just one block in memory, going to disk for another only when it becomes completely full or empty. An additional benefit of this approach is that by doing all the allocation from a single block of the bitmap, the disk blocks will be close together, thus minimizing disk-arm motion.

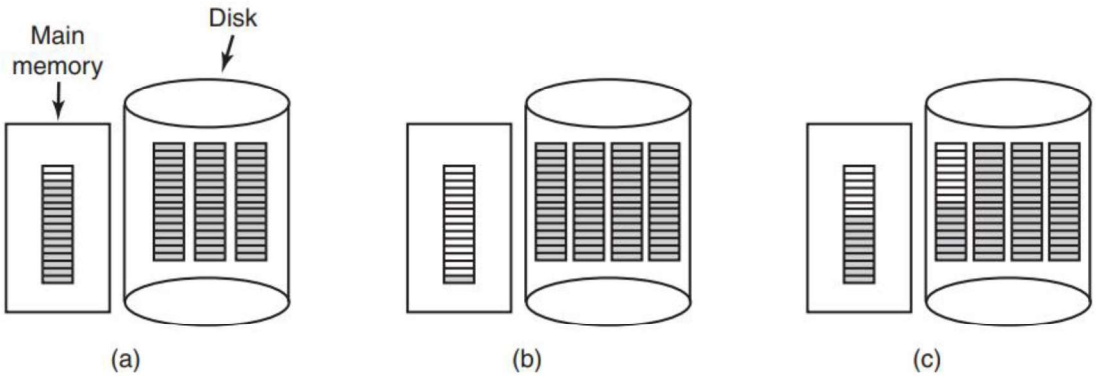

Figure 4-23. (a) An almost-full block of pointers to free disk blocks in memory and three blocks of pointers on disk. (b) Result of freeing a three-block file. (c) An alternative strategy for handling the three free blocks. The shaded entries represent pointers to free disk blocks.

Since the bitmap is a fixed-size data structure, if the kernel is (partially) paged, the bitmap can be put in virtual memory and have pages of it paged in as needed.

Disk Quotas

To prevent people from hogging too much disk space, multiuser operating systems often provide a mechanism for enforcing disk quotas. The idea is that the system administrator assigns each user a maximum allotment of files and blocks, and the operating system makes sure that the users do not exceed their quotas. A typical mechanism is described below.

When a user opens a file, the attributes and disk addresses are located and put into an open-file table in main memory. Among the attributes is an entry telling who the owner is. Any increases in the file's size will be charged to the owner's quota.

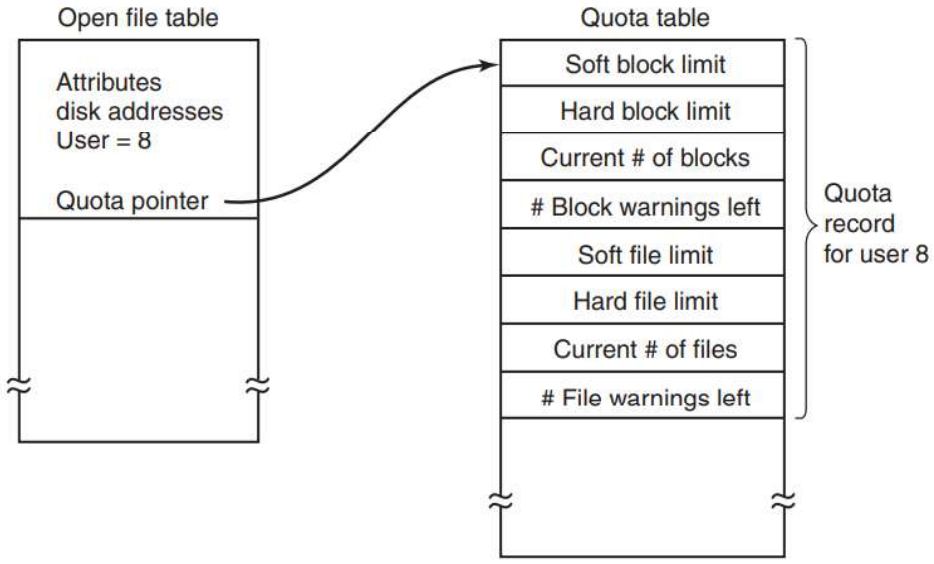

Figure 4-24. Quotas are kept track of on a per-user basis in a quota table.

If either limit has been violated, a warning is displayed, and the count of warnings remaining is reduced by one. If the count ever gets to zero, the user has ignored the warning one time too many, and is not permitted to log in. Getting permission to log in again will require some discussion with the system administrator.

This method has the property that users may go above their soft limits during a login session, provided they remove the excess before logging out. The hard limits may never be exceeded.

File-System Backups

Destruction of a file system is often a far greater disaster than destruction of a computer. If a computer is destroyed by fire, lightning surges, or a cup of coffee poured onto the keyboard, it is annoying and will cost money, but generally a replacement can be purchased with a minimum of fuss. Inexpensive personal computers can even be replaced within an hour by just going to a computer store (except at universities, where issuing a purchase order takes three committees, fiv e signatures, and 90 days).

If a computer's file system is irrevocably lost, whether due to hardware or software, restoring all the information will be difficult, time consuming, and in many cases, impossible. For the people whose programs, documents, tax records, customer files, databases, marketing plans, or other data are gone forever, the consequences can be catastrophic. While the file system cannot offer any protection against physical destruction of the equipment and media, it can help protect the information. It is pretty straightforward: make backups. But that is not quite as simple as it sounds. Let us take a look.

Most people do not think making backups of their files is worth the time and effort—until one fine day their disk abruptly dies, at which time most of them undergo a deathbed conversion. Companies, however, (usually) well understand the value of their data and generally do a backup at least once a day, often to tape. Modern tapes hold hundreds of gigabytes and cost pennies per gigabyte. Nevertheless, making backups is not quite as trivial as it sounds, so we will examine some of the related issues below.

Backups to tape are generally made to handle one of two potential problems:

1. **Recover from disaster.**
2. **Recover from stupidity**.

The first one covers getting the computer running again after a disk crash, fire, flood, or other natural catastrophe. In practice, these things do not happen very often, which is why many people do not bother with backups. These people also tend not to have fire insurance on their houses for the same reason.

The second reason is that users often accidentally remove files that they later need again. This problem occurs so often that when a file is ''removed'' in Windows, it is not deleted at all, but just moved to a special directory, the recycle bin, so it can be fished out and restored easily later. Backups take this principle further and allow files that were removed days, even weeks, ago to be restored from old backup tapes

Second, it is wasteful to back up files that have not changed since the previous backup, which leads to the idea of incremental dumps. The simplest form of **incremental dumping** is to make a complete dump (backup) periodically, say weekly or monthly, and to make a daily dump of only those files that have been modified since the last full dump. Even better is to dump only those files that have changed since they were last dumped. While this scheme minimizes dumping time, it makes recovery more complicated, because first the most recent full dump has to be restored, followed by all the incremental dumps in reverse order. To ease recovery, more sophisticated incremental dumping schemes are often used.

Tw o strategies can be used for dumping a disk to a backup disk: **a physical dump or a logical dump**. A physical dump starts at block 0 of the disk, writes all the disk blocks onto the output disk in order, and stops when it has copied the last one. Such a program is so simple that it can probably be made 100% bug free, something that can probably not be said about any other useful program.

A logical dump starts at one or more specified directories and recursively dumps all files and directories found there that have changed since some given base date (e.g., the last backup for an incremental dump or system installation

for a full dump). Thus, in a logical dump, the dump disk gets a series of carefully identified directories and files, which makes it easy to restore a specific file or directory upon request.

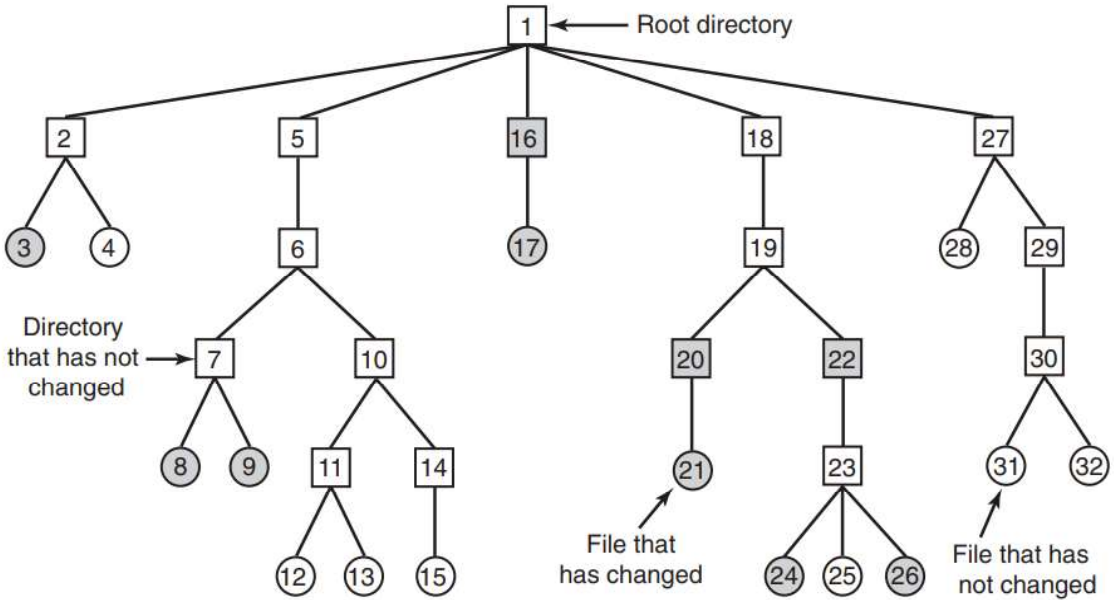

Figure 4-25. A file system to be dumped. The squares are directories and the circles are files. The shaded items have been modified since the last dump. Each directory and file is labeled by its i-node number.

At the end of phase 1, all modified files and all directories have been marked in the bitmap, as shown (by shading) in Fig. 4-26(a). Phase 2 conceptually recursively walks the tree again, unmarking any directories that have no modified files or directories in them or under them. This phase leaves the bitmap as shown in Fig. 4-26(b). Note that directories 10, 11, 14, 27, 29, and 30 are now unmarked because they contain nothing under them that has been modified. They will not be dumped. By way of contrast, directories 5 and 6 will be dumped even though they themselves have not been modified because they will be needed to restore today's changes to a fresh machine. For efficiency, phases 1 and 2 can be combined in one tree walk.

At this point it is known which directories and files must be dumped. These are the ones that are marked in Fig. 4-26(b). Phase 3 consists of scanning the i-nodes in numerical order and dumping all the directories that are **marked for dumping.**

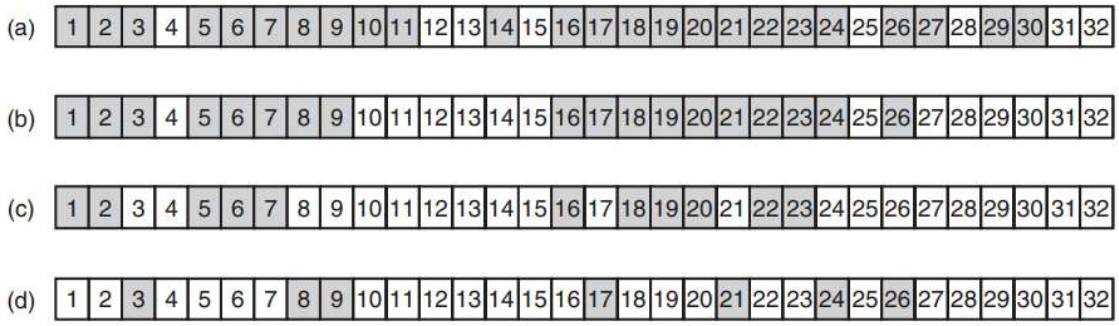

Figure 4-26. Bitmaps used by the logical dumping algorithm.

These are shown in Fig. 4-26(c). Each directory is prefixed by the directory's attributes (owner, times, etc.) so that they can be restored. Finally, in phase 4, the files marked in Fig. 4-26(d) are also dumped, again prefixed by their attributes. This completes the dump.

Still another issue is the fact that UNIX files may contain holes. It is legal to open a file, write a few bytes, then seek to a distant file offset and write a few more bytes. The blocks in between are not part of the file and should not be dumped and must not be restored. Core files often have a hole of hundreds of megabytes between the data segment and the stack. If not handled properly, each restored core file will fill this area with zeros and thus be the same size as the virtual address space (e.g., 2 power power32 bytes, or worse yet, 2 power 64 bytes).

Finally, special files, named pipes, and the like (anything that is not a real file) should never be dumped, no matter in which directory they may occur (they need not be confined to /dev). For more information about file-system backups, see Chervenak et al., (1998) and Zwicky (1991).

File-System Consistency

Another area where reliability is an issue is file-system consistency. Many file systems read blocks, modify them, and write them out later. If the system crashes before all the modified blocks have been written out, the file system can be left in an inconsistent state. This problem is especially critical if some of the blocks that have not been written out are i-node blocks, directory blocks, or blocks containing the free list.

To deal with inconsistent file systems, most computers have a utility program that checks file-system consistency. For example, UNIX has fsck; Windows has sfc (and others). This utility can be run whenever the system is booted, especially after a crash. The description below tells how fsck works. Sfc is somewhat different because it works on a different file system, but the general principle of using the file system's inherent redundancy to repair it is still valid. All file-system checkers verify each file system (disk partition) independently of the other ones.

If the file system is consistent, each block will have a 1 either in the first table or in the second table, as illustrated in Fig. 4-27(a). However, as a result of a crash, the tables might look like Fig. 4-27(b), in which block 2 does not occur in either table. It will be reported as being a **missing block**. While missing blocks do no real harm, they waste space and thus reduce the capacity of the disk. The solution to missing blocks is straightforward: the file system checker just adds them to the free list.

Another situation that might occur is that of Fig. 4-27(c). Here we see a block, number 4, that occurs twice in the free list. (Duplicates can occur only if the free list is really a list; with a bitmap it is impossible.) The solution here is also simple: rebuild the free list.

The worst thing that can happen is that the same data block is present in two or more files, as shown in Fig. 4-27(d) with block 5. If either of these files is removed, block 5 will be put on the free list, leading to a

situation in which the same block is both in use and free at the same time. If both files are removed, the block will be put onto the free list twice.

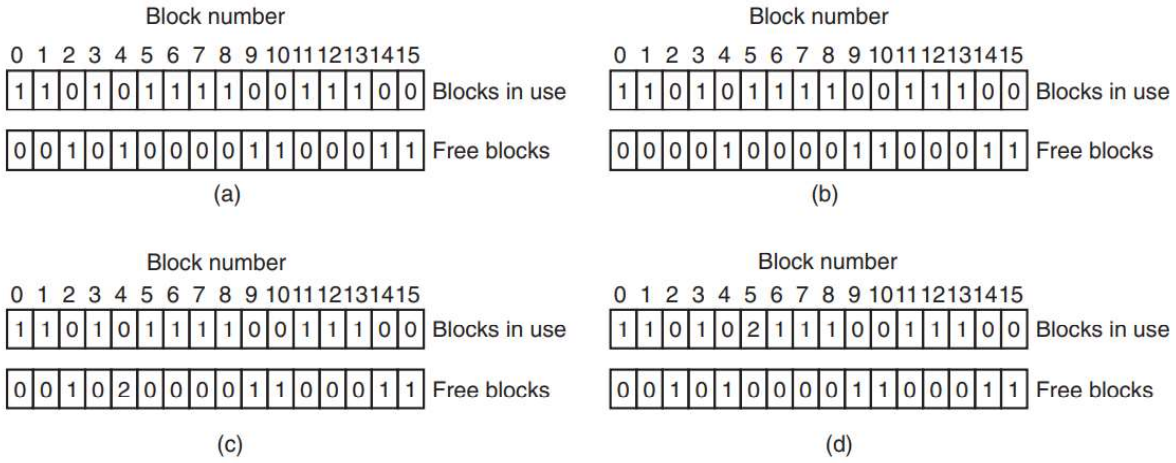

Figure 4-27. File-system states. (a) Consistent. (b) Missing block. (c) Duplicate block in free list. (d) Duplicate data block.

The appropriate action for the file-system checker to take is to allocate a free block, copy the contents of block 5 into it, and insert the copy into one of the files. In this way, the information content of the files is unchanged (although almost assuredly one is garbled), but the file-system structure is at least made consistent. The error should be reported, to allow the user to inspect the damage.

In addition to checking to see that each block is properly accounted for, the file-system checker also checks the directory system. It, too, uses a table of counters, but these are per file, rather than per block. It starts at the root directory and recursively descends the tree, inspecting each directory in the file system. For every i-node in every directory, it increments a counter for that file's usage count. Remember that due to hard links, a file may appear in two or more directories. Symbolic links do not count and do not cause the counter for the target file to be incremented.

When the checker is all done, it has a list, indexed by i-node number, telling how many directories contain each file. It then compares these numbers with the link counts stored in the i-nodes themselves. These counts start at 1 when a file is created and are incremented each time a (hard) link is made to the file. In a consistent file system, both counts will agree. However, two kinds of errors can occur: the link count in the i-node can be too high or it can be too low.

If the link count is higher than the number of directory entries, then even if all the files are removed from the directories, the count will still be nonzero and the i-node will not be removed. This error is not serious, but it wastes space on the disk with files that are not in any directory. It should be fixed by setting the link count in the i-node to the correct value.

The other error is potentially catastrophic. If two directory entries are linked to a file, but the i-node says that there is only one, when either directory entry is removed, the i-node count will go to zero. When an i-node count goes to zero, the

file system marks it as unused and releases all of its blocks. This action will result in one of the directories now pointing to an unused i-node, whose blocks may soon be assigned to other files. Again, the solution is just to force the link count in the inode to the actual number of directory entries.

These two operations, checking blocks and checking directories, are often integrated for efficiency reasons (i.e., only one pass over the i-nodes is required). Other checks are also possible. For example, directories have a definite format, with i-node numbers and ASCII names. If an i-node number is larger than the number of i-nodes on the disk, the directory has been damaged.

CHAPTER -3

Secondary-Storage Structure:

1. Overview of disk structure,
2. and attachment,
3. Disk scheduling,
4. RAID structure,
5. Stable storage implementation.

1). OVERVIEW OF MASS-STORAGE STRUCTURE

MAGNETIC DISKS:

Magnetic disks provide the bulk of secondary storage for modern computer systems. Conceptually, disks are relatively simple (Figure 4.1). Each disk **platter** has a flat circular shape, like a CD. Common platter diameters range from 1.8 to 3.5 inches. The two surfaces of a platter are covered with a magnetic material. We store information by recording it magnetically on the platters.

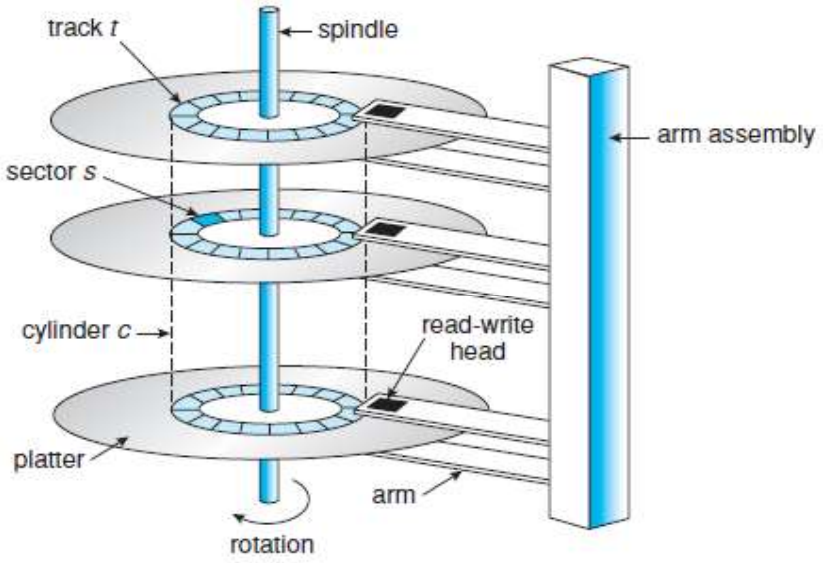

FIGURE 4.1: MOVING-HEAD DISK MECHANISM

A read–write head "flies" just above each surface of every platter. The heads are attached to a **disk arm** that moves all the heads as a unit. The surface of a platter is logically divided into circular **tracks**, which are subdivided into **sectors**. The set of tracks that are at one arm position makes up a **cylinder**. There may be thousands of concentric cylinders in a disk drive, and each track may contain hundreds of sectors. The storage capacity of common disk drives is measured in gigabytes.

When the disk is in use, a drive motor spins it at high speed. Most drives rotate 60 to 250 times per second, specified in terms of rotations per minute **(RPM)**. Common drives spin at 5,400, 7,200, 10,000, and 15,000 RPM. Disk speed has two parts.

The **transfer rate** is the rate at which data flow between the drive and the computer. The **positioning time**, or **random-access time**, consists of two parts: the time necessary to move the disk arm to the desired cylinder, called the **seek time**, and the time necessary for the desired sector to rotate to the disk head, called the **rotational latency**.

Typical disks can transfer several megabytes of data per second, and they have seek times and rotational latencies of several milliseconds.

Because the disk head flies on an extremely thin cushion of air (measured in microns), there is a danger that the head will make contact with the disk surface. Although the disk platters are coated with a thin protective layer, the head will sometimes damage the magnetic surface. This accident is called a **head crash**. A head crash normally cannot be repaired; the entire disk must be replaced.

A disk can be **removable**, allowing different disks to be mounted as needed. Removable magnetic disks generally consist of one platter, held in a plastic case to prevent damage while not in the disk drive. Other forms of removable disks include CDs, DVDs, and Blu-ray discs as well as removable flash-memory devices known as **flash drives** (which are a type of solid-state drive).

A disk drive is attached to a computer by a set of wires called an **I/O bus**. Several kinds of buses are available, including **advanced technology attachment (ATA)**, **serial ATA (SATA)**, **eSATA**, **universal serial bus (USB)**, and **fibre channel (FC)**. The data transfers on a bus are carried out by special electronic processors called **controllers**. The **host controller** is the controller at the computer end of the bus. A **disk controller** is built into each disk drive.

SOLID-STATE DISKS:

Sometimes old technologies are used in new ways as economics change or the technologies evolve. An example is the growing importance of **solid-state disks**, or **SSDs**. Simply described, an SSD is nonvolatile memory that is used like a hard drive. There are many variations of this technology, from DRAM with a battery to allow it to maintain its state in a power failure through flash-memory technologies like single-level cell (SLC) and multilevel cell (MLC) chips.

SSDs have the same characteristics as traditional hard disks but can be more reliable because they have no moving parts and faster because they have no seek time or latency. In addition, they consume less power.

However, they are more expensive per megabyte than traditional hard disks, have less capacity than the larger hard disks, and may have shorter life spans than hard disks, so their uses are somewhat limited. One use for SSDs is in storage arrays, where they hold file-system metadata that require high performance. SSDs are also used in some laptop computers to make them smaller, faster, and more energy-efficient.

MAGNETIC TAPES:

Magnetic tape was used as an early secondary-storage medium. Although it is relatively permanent and can hold large quantities of data, its access time is slow compared with that of main memory and magnetic disk. In addition, random access to magnetic tape is about a thousand times slower than random access to magnetic disk, so tapes are not very useful for secondary storage.

Tapes are used mainly for backup, for storage of infrequently used information, and as a medium for transferring information from one system to another. A tape is kept in a spool and is wound or rewound past a read–write head.

Moving to the correct spot on a tape can take minutes, but once positioned, tape drives can write data at speeds comparable to disk drives. Tape capacities vary greatly, depending

on the particular kind of tape drive, with current capacities exceeding several terabytes. Some tapes have built-in compression that can more than double the effective storage. Tapes and their drivers are usually categorized by width, including 4, 8, and 19 millimeters and 1/4 and 1/2 inch.

DISK STRUCTURE:

Modern magnetic disk drives are addressed as large one-dimensional arrays of **logical blocks**, where the logical block is the smallest unit of transfer. The size of a logical block is usually 512 bytes, although some disks can be **low-level formatted** to have a different logical block size, such as 1,024 bytes.

The one-dimensional array of logical blocks is mapped onto the sectors of the disk sequentially. Sector 0 is the first sector of the first track on the outermost cylinder. The mapping proceeds in order through that track, then through the rest of the tracks in that cylinder, and then through the rest of the cylinders from outermost to innermost. By using this mapping, we can—at least in theory—convert a logical block number into an old-style disk address that consists of a cylinder number, a track number within that cylinder, and a sector number within that track. In practice, it is difficult to perform this translation, for two reasons.

First, most disks have some defective sectors, but the mapping hides this by substituting spare sectors from elsewhere on the disk. Second, the number of sectors per track is not a constant on some drives.

2)DISK ATTACHMENT:

Computers access disk storage in two ways. One way is via I/O ports (or **host-attached storage**); this is common on small systems. The other way is via a remote host in a distributed file system; this is referred to as **network-attached storage**.

1. Host-Attached Storage:

Host-attached storage is storage accessed through local I/O ports. These ports use several technologies. The typical desktop PC uses an I/O bus architecture called IDE or ATA. This architecture supports a maximum of two drives per I/O bus. A newer, similar protocol that has simplified cabling is SATA.

High-end workstations and servers generally use more sophisticated I/O architectures such as fibre channel (FC), a high-speed serial architecture that can operate over optical fiber or over a four-conductor copper cable. It has two variants. One is a large switched fabric having a 24-bit address space. This variant is expected to dominate in the future and is the basis of **storage-area networks (SANs).** The other FC variant is an **arbitrated loop (FC-AL)** that can address 126 devices (drives and controllers).

A wide variety of storage devices are suitable for use as host-attached storage. Among these are hard disk drives, RAID arrays, and CD, DVD, and tape drives. The I/O commands that initiate data transfers to a host-attached storage device are reads andwrites of logical data blocks directed to specifically identified storage units (such as bus ID or target logical unit).

2. Network-Attached Storage:

A network-attached storage (NAS) device is a special-purpose storage system that is accessed remotely over a data network (Figure 4.2). Clients access network-attached storage via a remote-procedure-call interface such as NFS for UNIX systems or CIFS for Windows machines.

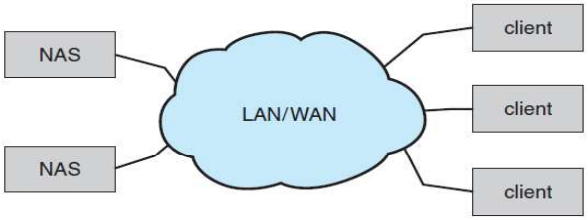

FIGURE 4.2: NETWORK-ATTACHED STORAGE

The remote procedure calls (RPCs) are carried via TCP or UDP over an IP network—usually the same local area network (LAN) that carries all data traffic to the clients. Thus, it may be easiest to think of NAS as simply another storage- access protocol. The network attached storage unit is usually implemented as a RAID array with software that implements the RPC interface.

Network-attached storage provides a convenient way for all the computers on a LAN to share a pool of storage with the same ease of naming and access enjoyed with local host-attached storage. However, it tends to be less efficient and have lower performance than some direct-attached storage options. **iSCSI** is the latest network-attached storage protocol. In essence, it uses the IP network protocol to carry the SCSI protocol.

3. Storage-Area Network:

One drawback of network-attached storage systems is that the storage I/O operations consume bandwidth on the data network, thereby increasing the latency of network communication.

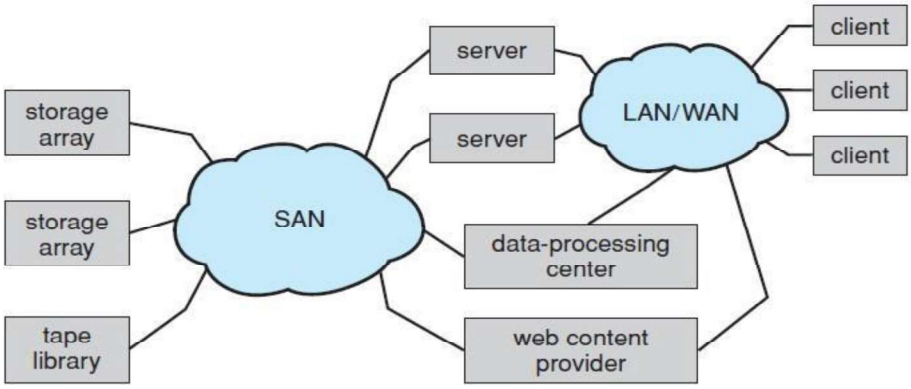

FIGURE 4.3: STORAGE-AREA NETWORK

A storage-area network (SAN) is a private network (using storage protocols rather than networking protocols) connecting servers and storage units, as shown in Figure 4.3. The power of a SAN lies in its flexibility. Multiple hosts and multiple storage arrays can attach to the same SAN, and storage can be dynamically allocated to hosts. A SAN switch allows or prohibits access between the hosts and the storage.

3)DISK SCHEDULING:

One of the responsibilities of the operating system is to use the hardware efficiently. For the disk drives, meeting this responsibility entails having fast access time and large disk bandwidth.

The **seek time** is the time for the disk arm to move the heads to the cylinder containing the desired sector. The **rotational latency** is the additional time for the disk to rotate the desired sector to the disk head. The disk **bandwidth** is the total number of bytes transferred, divided by the total time between the first request for service and the completion of the last transfer. We can improve both the access time and the bandwidth by managing the order in which disk I/O requests are serviced.

Whenever a process needs I/O to or from the disk, it issues a system call to the operating system. The request specifies several pieces of information:

- Whether this operation is input or output
- What the disk address for the transfer is
- What the memory address for the transfer is
- What the number of sectors to be transferred is

If the desired disk drive and controller are available, the request can be serviced immediately. If the drive or controller is busy, any new requests for service will be placed in the queue of pending requests for that drive.

For a multiprogramming system with many processes, the disk queue may often have several pending requests. Thus, when one request is completed, the operating system chooses which pending request to service next. The Operating system make this choice with one of several disk-scheduling algorithms.

1. FCFS SCHEDULING:

The simplest form of disk scheduling is, of course, the first-come, first-served (FCFS) algorithm. This algorithm is intrinsically fair, but it generally does not provide the fastest service. Consider, for example, a disk queue with requests for I/O to blocks on cylinders ***98, 183, 37, 122, 14, 124, 65, 67***, in that order.

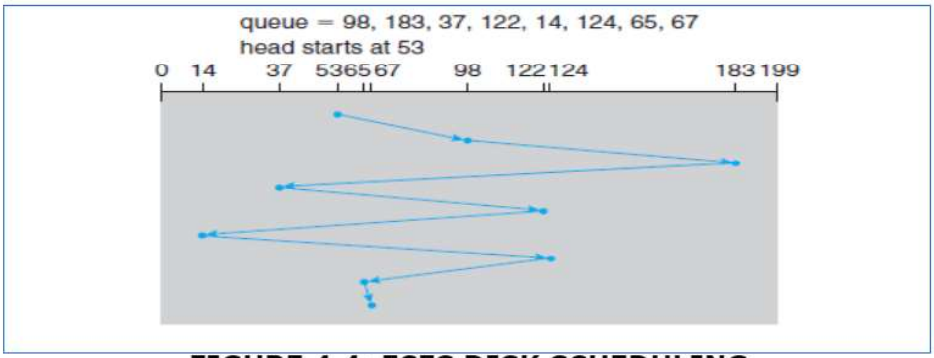

FIGURE 4.4: FCFS DISK SCHEDULING

If the disk head is initially at cylinder 53, it will first move from 53 to 98, then to 183, 37, 122, 14, 124, 65, and finally to 67, for a total head movement of 640 cylinders. This schedule is diagrammed in Figure 4.4. ***The wild swing from 122 to 14 and then back to 124 illustrates the problem with this schedule.***

2. SSTF SCHEDULING:

It seems reasonable to service all the requests close to the current head position before moving the head far away to service other requests. This assumption is the basis for the **shortest-seek-time-first (SSTF) algorithm**. The SSTF algorithm selects the request with the least seek time from the current head position. In other words, SSTF chooses the pending request closest to the current head position.

For our example request queue, the closest request to the initial head position (53) is at cylinder 65. Once we are at cylinder 65, the next closest request is at cylinder 67. From there, the request at cylinder 37 is closer than the one at 98, so 37 is served next. Continuing, we service the request at cylinder 14, then 98, 122, 124, and finally 183 (Figure 4.5).

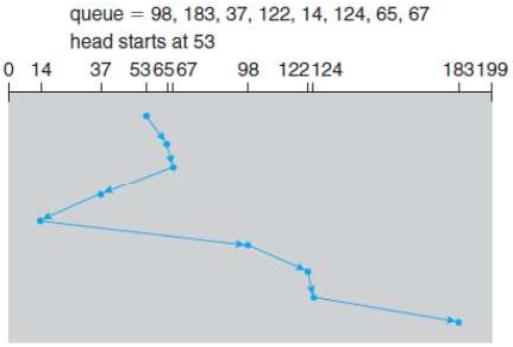

FIGURE 4.5: SSTF DISK SCHEDULING

This scheduling method results in a total head movement of only 236 cylinders—little more than one-third of the distance needed for FCFS scheduling of this request queue. Clearly, this algorithm gives a substantial improvement in performance. Although the SSTF algorithm is a substantial improvement over the FCFS algorithm, it is not optimal.

3. SCAN SCHEDULING:

In the **SCAN algorithm**, the disk arm starts at one end of the disk and moves toward the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk. At the other end, the direction of head movement is reversed, and servicing continues.

The head continuously scans back and forth across the disk. The SCAN algorithm is sometimes called the **elevator algorithm**, since the disk arm behaves just like an elevator in a building, first servicing all the requests going up and then reversing to service requests the other way. Let's return to our example to illustrate. Before applying SCAN to schedule the requests on cylinders 98, 183, 37, 122, 14, 124, 65, and 67, we need to know the direction of head movement in addition to the head's current position.

Assuming that the disk arm is moving toward 0 and that the initial head position is again 53, the head will next service 37 and then 14. At cylinder 0, the arm will reverse and will move toward the other end of the disk, servicing the requests at 65, 67, 98, 122, 124, and 183 (Figure 4.6). If a request arrives in the queue just in front of the head, it will be serviced almost immediately; a request arriving just behind the head will have to wait until the arm moves to the end of the disk, reverses direction, and comes back

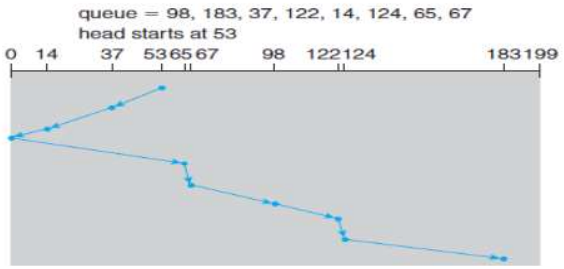

FIGURE 4.6: SCAN DISK SCHEDULING

4. C-SCAN SCHEDULING:

Circular SCAN (C-SCAN) scheduling is a variant of SCAN designed to provide a more uniform wait time. Like SCAN, C-SCAN moves the head from one end of the disk to the other, servicing requests along the way. When the head reaches the other end, however, it immediately returns to the beginning of the disk without servicing any requests on the return trip (Figure 4.7). The C-SCAN scheduling algorithm essentially treats the cylinders as a circular list that wraps around from the final cylinder to the first one.

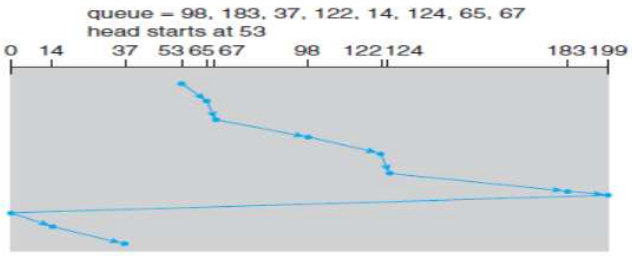

FIGURE 4.7: C-SCAN DISK SCHEDULING

5. LOOK SCHEDULING:

As we described them, both SCAN and C-SCAN move the disk arm across the full width of the disk. In practice, neither algorithm is often implemented this way. More commonly, the arm goes only as far as the final request in each direction.

Then, it reverses direction immediately, without going all the way to the end of the disk. Versions of SCAN and C-SCAN that follow this pattern are called **LOOK** and **C-LOOK scheduling**, because they ***look*** for a request before continuing to move in a given direction (Figure 4.8).

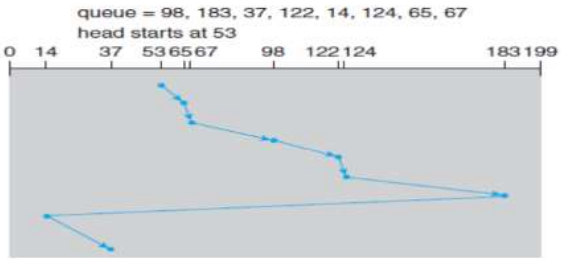

FIGURE 4.8: C-LOOK DISK SCHEDULING

SWAP-SPACE MANAGEMENT:

Swap-space management is another low-level task of the operating system. Virtual memory uses disk space as an extension of main memory. Since disk access is much slower than memory access, using swap space significantly decreases system performance. The main goal for the design and implementation of swap space is to provide the best throughput for the virtual memory system.

SWAP-SPACE USE:

Swap space is used in various ways by different operating systems, depending on the memory-management algorithms in use. For instance, systems that implement swapping may use swap space to hold an entire process image, including the code and data segments.

Paging systems may simply store pages that have been pushed out of main memory. The amount of swap space needed on a system can therefore vary from a few megabytes of disk space to gigabytes, depending on the amount of physical memory, the amount of virtual memory it is backing, and the way in which the virtual memory is used. Note that it may be safer to overestimate than to underestimate the amount of swap space required, because if a system runs out of swap space it may be forced to abort processes or may crash entirely.

Overestimation wastes disk space that could otherwise be used for files, but it does no other harm. Some systems recommend the amount to be set aside for swap space.

SWAP-SPACE LOCATION: A swap space can reside in one of two places:

1) *It can be carved out of the normal file system*
2) *It can be in a separate disk partition.*

If the swap space is simply a large file within the file system, normal file-system routines can be used to create it, name it, and allocate its space. This approach, though easy to implement, is inefficient because navigating the directory structure and the disk allocation data structures takes time and (possibly) extra disk accesses. External fragmentation can greatly increase swapping times by forcing multiple seeks during reading or writing of a process image.

Alternatively, swap space can be created in a separate **raw partition**. No file system or directory structure is placed in this space. Rather, a separate swap-space storage manager is used to allocate and deallocate the blocks from the raw partition.

This manager uses algorithms optimized for speed rather than for storage efficiency, because swap space is accessed much more frequently than file systems (when it is used). Internal fragmentation may increase which is acceptable. Some operating systems are flexible and can swap both in raw partitions and in file- system space.

4)RAID STRUCTURE:

Disk drives have continued to get smaller and cheaper, so it is now economically feasible to attach many disks to a computer system. Having a large number of disks in a system presents opportunities for improving the rate at which data can be read or written, if the disks are operated in parallel.

Furthermore, this setup offers the potential for improving the reliability of data storage, because redundant information can be stored on multiple disks. Thus, failure of one

disk does not lead to loss of data. A variety of disk-organization techniques, collectively called **redundant arrays of independent disks (RAID)**, are commonly used to address the performance and reliability issues.

In the past, RAIDs composed of small, cheap disks were viewed as a cost- effective alternative to large, expensive disks. Today, RAIDs are used for their higher reliability and higher data-transfer rate, rather than for economic reasons. Hence, the ***I*** in ***RAID,*** which once stood for "inexpensive," now stands for "independent."

Improvement Of Reliability Via Redundancy:

Let's first consider reliability of RAIDs. The chance that some disk out of a set of *N* disks will fail is much higher than the chance that a specific single disk will fail.

The solution to the problem of reliability is to introduce **redundancy**; we store extra information that is not normally needed but that can be used in the event of failure of a disk to rebuild the lost information. Thus, even if a disk fails, data are not lost.

The simplest (but most expensive) approach to introducing redundancy is to duplicate every disk. This technique is called **mirroring**. With mirroring, a logical disk consists of two physical disks, and every write is carried out on both disks. The result is called a **mirrored volume**. If one of the disks in the volume fails, the data can be read from the other. Data will be lost only if the second disk fails before the first failed disk is replaced.

The mean time to failure of a mirrored volume—where failure is the loss of data—depends on two factors. One is the mean time to failure of the individual disks. The other is the **mean time to repair**, which is the time it takes (on average) to replace a failed disk and to restore the data on it.

Improvement In Performance Via Parallelism:

Now let's consider how parallel access to multiple disks improves performance. With disk mirroring, the rate at which read requests can be handled is doubled, since read requests can be sent to either disk (as long as both disks in a pair are functional, as is almost always the case).

The transfer rate of each read is the same as in a single-disk system, but the number of reads per unit time has doubled. With multiple disks, we can improve the transfer rate as well (or instead) by striping data across the disks.

In its simplest form, **data striping** consists of splitting the bits of each byte across multiple disks; such striping is called **bit-level striping**.

In **block-level striping**, for instance, blocks of a file are striped across multiple disks; with *n* disks, block *i* of a file goes to disk (*i* mod *n*) + 1. Other levels of striping, such as bytes of a sector or sectors of a block, also are possible. Block- level striping is the most common.

Parallelism in a disk system, as achieved through striping, has two main goals:

- Increase the throughput of multiple small accesses by load balancing.
- Reduce the response time of large accesses.

RAID LEVELS:

RAID Levels

Mirroring provides high reliability, but it is expensive. Striping provides high data-transfer rates, but it does not improve reliability. Numerous schemes to provide redundancy at lower cost by using disk striping combined with "parity" bits have been proposed. These schemes have different cost–performance trade-offs and are classified according to levels called **RAID levels**.

We describe the various levels here; Figure 4.9 shows them pictorially (in the figure, *P* indicates error-correcting bits and *C* indicates a second copy of the data). In all cases depicted in the figure, four disks' worth of data are stored, and the extra disks are used to store redundant information for failure recovery.

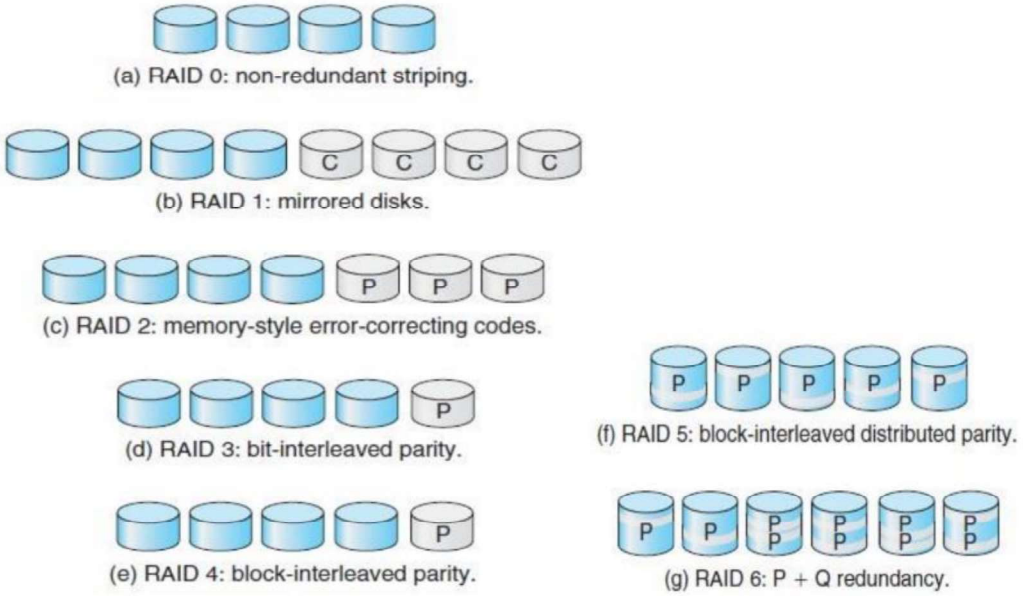

Figure 4.9: RAID LEVELS

- **RAID level 0**. RAID level 0 refers to disk arrays with striping at the level of blocks but without any redundancy (such as mirroring or parity bits), as shown in Figure 4.9(a).
- **RAID level 1**. RAID level 1 refers to disk mirroring. Figure 4.9(b) shows a mirrored organization.
- **RAID level 2**. RAID level 2 is also known as memory-style error-correcting code (ECC) organization. Memory systems have long detected certain errors by using parity bits.
- **RAID level 3**. RAID level 3, or bit-interleaved parity organization, improves on level 2 by taking into account the fact that, unlike memory systems, disk controllers can detect whether a sector has been read correctly, so a single parity bit can be used for error correction as well as for detection.
- **RAID level 4**. RAID level 4, or block-interleaved parity organization, uses block-level striping, as in RAID 0, and in addition keeps a parity block on a separate disk for corresponding blocks from *N* other disks. This scheme is diagrammed in Figure 4.9(e). If one of the disks fails, the parity block can be used with the corresponding blocks from the other disks to restore the blocks of the failed disk.

- **RAID level 5**. RAID level 5, or block-interleaved distributed parity, differs from level 4 in that it spreads data and parity among all *N*+1 disks, rather than storing data in *N* disks and parity in one disk.

- **RAID level 6**. RAID level 6, also called the **P + Q redundancy scheme**, is much like RAID level 5 but stores extra redundant information to guard against multiple disk failures.
- **RAID levels 0 + 1 and1 + 0.**
  - RAID level 0 + 1 refers to a combination of RAID levels 0 and 1. RAID 0 provides the performance, while RAID 1 provides the reliability. Generally, this level provides better performance than RAID 5.
  - Another RAID option that is becoming available commercially is RAID level 1+ 0, in which disks are mirrored in pairs and then the resulting mirrored pairs are striped. This scheme has some theoretical advantages over RAID 0 + 1.

For example, if a single disk fails in RAID 0 + 1, an entire stripe is inaccessible, leaving only the other stripe. With a failure in RAID 1 + 0, a single disk is unavailable, but the disk that mirrors it is still available, as are all the rest of the disks (Figure 4.10).

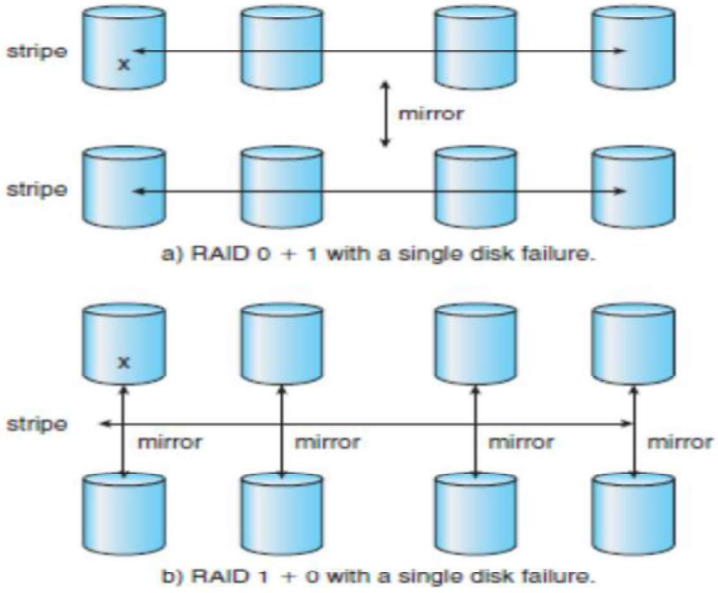

FIGURE 4.10: RAID 0 + 1 AND 1 + 0

STABLE-STORAGE IMPLEMENTATION:

By definition, information residing in stable storage is never lost. To implement such storage, we need to replicate the required information. on multiple storage devices (usually disks) with independent failure modes.

We also need to coordinate the writing of updates in a way that guarantees that a failure during an update will not leave all the copies in a damaged state and that, when we are recovering from a failure, we can force all copies to a consistent and correct value, even if another failure occurs during the recovery.

A disk writes results in one of three outcomes:

1. **Successful completion**. The data were written correctly on disk.
2. **Partial failure**. A failure occurred in the midst of transfer, so only some of the sectors were written with the new data, and the sector being written during the failure may have been corrupted.

3. **Total failure**. The failure occurred before the disk write started, so the previous data values on the disk remain intact.

Whenever a failure occurs during writing of a block, the system needs to detect it and invoke a recovery procedure to restore the block to a consistent state. To do that, the system must maintain two physical blocks for each logical block. An output operation is executed as follows:

1. Write the information onto the first physical block.
2. When the first write completes successfully, write the same information onto the second physical block.
3. Declare the operation complete only after the second write completes successfully.

Unit-5

Chapter-1

PROTECTION

1. GOALS OF PROTECTION

- Obviously to prevent malicious misuse of the system by users or programs. See chapter 7 for a more thorough coverage of this goal.
- To ensure that each shared resource is used only in accordance with system *policies,* which may be set either by system designers or by system administrators.
- To ensure that errant programs cause the minimal amount of damage possible.
- Note that protection systems only provide the *mechanisms* for enforcing policies and ensuring reliable systems. It is up to administrators and users to implement those mechanisms effectively.

2 PRINCIPLES OF PROTECTION

- The **principle of least privilege** dictates that programs, users, and systems be given just enough privileges to perform their tasks.
- This ensures that failures do the least amount of harm and allow the least of harm to be done.
- For example, if a program needs special privileges to perform a task, it is better to make it a SGID program with group ownership of "network" or "backup" or some other pseudo group, rather than SUID with root ownership. This limits the amount of damage that can occur if something goes wrong.
- Typically each user is given their own account, and has only enough privilege to modify their own files.
- The root account should not be used for normal day to day activities - The System Administrator should also have an ordinary account, and reserve use of the root account for only those tasks which need the root privileges

3 DOMAIN OF PROTECTION

- A computer can be viewed as a collection of *processes* and *objects* (both HW & SW).
- The **need to know principle** states that a process should only have access to those objects it needs to accomplish its task, and furthermore only in the modes for which it needs access and only during the time frame when it needs access.
- The modes available for a particular object may depend upon its type.

3.1 Domain Structure

- A **protection domain** specifies the resources that a process may access.

- Each domain defines a set of objects and the types of operations that may be invoked on each object.
- An **access right** is the ability to execute an operation on an object.
- A domain is defined as a set of < object, { access right set } > pairs, as shown below. Note that some domains may be disjoint while others overlap.
- The association between a process and a domain may be *static* or *dynamic.*
    - If the association is static, then the need-to-know principle requires a way of changing the contents of the domain dynamically.
    - If the association is dynamic, then there needs to be a mechanism for **domain switching.**

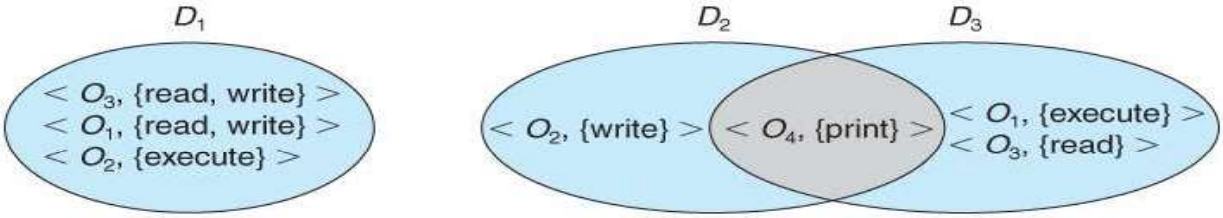

Figure - System with three protection domains.

- Domains may be realized in different fashions - as users, or as processes, or as procedures. E.g. if each user corresponds to a domain, then that domain defines the access of that user, and changing domains involves changing user ID.

3.2 An Example: UNIX

- UNIX associates domains with users.
- Certain programs operate with the SUID bit set, which effectively changes the user ID, and therefore the access domain, while the program is running. (and similarly for the SGID bit.) Unfortunately this has some potential for abuse.
- An alternative used on some systems is to place privileged programs in special directories, so that they attain the identity of the directory owner when they run. This prevents crackers from placing SUID programs in random directories around the system.

3.3 An Example: MULTICS

- The MULTICS system uses a complex system of rings, each corresponding to a different protection domain, as shown below:

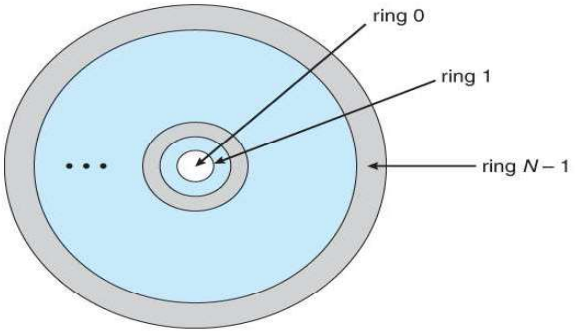

Figure - MULTICS ring structure.

- Rings are numbered from 0 to 7, with outer rings having a subset of the privileges of the inner rings.
- Each file is a memory segment, and each segment description includes an entry that indicates the ring number associated with that segment, as well as read, write, and execute privileges.
- Each process runs in a ring, according to the *current-ring-number,* a counter associated with each process.
- A process operating in one ring can only access segments associated with higher (farther out) rings, and then only according to the access bits. Processes cannot access segments associated with lower rings.
- Domain switching is achieved by a process in one ring calling upon a process operating in a lower ring, which is controlled by several factors stored with each segment descriptor:
  - An **access bracket**, defined by integers b1 <= b2.
  - A **limit** b3 > b2
  - A **list of gates,** identifying the entry points at which the segments may be called.
- If a process operating in ring i calls a segment whose bracket is such that b1 <= i <= b2, then the call succeeds and the process remains in ring i.
- Otherwise a trap to the OS occurs.

4 ACCESS MATRIX

- The model of protection that we have been discussing can be viewed as an **access matrix,** in which columns represent different system resources and rows represent different protection domains. Entries within the matrix indicate what access that domain has to that resource.

object / domain	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

Figure - Access matrix.

- Domain switching can be easily supported under this model, simply by providing "switch" access to other domains:

object / domain	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch
D_3		read	execute					
D_4	read write		read write		switch			

Figure - Access matrix of Figure 6.3 with domains as objects.

- The ability to **copy** rights is denoted by an asterisk, indicating that processes in that domain have the right to copy that access within the same column, i.e. for the same object. There are two important variations:
  - If the asterisk is removed from the original access right, then the right is **transferred,** rather than being copied. This may be termed a **transfer** right as opposed to a **copy** right.
  - If only the right and not the asterisk is copied, then the access right is added to the new domain, but it may not be propagated further. That is the new domain does not also receive the right to copy the access. This may be termed a **limited copy** right, as shown in Figure 6.5 below:

object / domain	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute		

(a)

object / domain	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute	read	

(b)

Figure – Access matrix with *copy* rights.

- The **owner** right adds the privilege of adding new rights or removing existing ones:

object / domain	F_1	F_2	F_3
D_1	owner execute		write
D_2		read* owner	read* owner write
D_3	execute		

(a)

object / domain	F_1	F_2	F_3
D_1	owner execute		write
D_2		owner read* write*	read* owner write
D_3		write	write

(b)

Figure - Access matrix with *owner* rights.

- Copy and owner rights only allow the modification of rights within a column. The addition of **control rights**, which only apply to domain objects, allow a process operating in one domain to affect the rights available in other domains. For example in the table below, a process operating in domain D2 has the right to control any of the rights in domain D4.

object / domain	F_1	F_2	F_3	laser printer	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch control
D_3		read	execute					
D_4	write		write		switch			

Figure - Modified access matrix of Figure 6.4

4.1 IMPLEMENTATION OF ACCESS MATRIX

4.1.1 Global Table

- The simplest approach is one big global table with < domain, object, rights > entries.
- Unfortunately this table is very large (even if sparse) and so cannot be kept in memory (without invoking virtual memory techniques.)
- There is also no good way to specify groupings - If everyone has access to some resource, then it still needs a separate entry for every domain.

4.1.2 Access Lists for Objects

- Each column of the table can be kept as a list of the access rights for that particular object, discarding blank entries.
- For efficiency a separate list of default access rights can also be kept, and checked first.

4.1.3 Capability Lists for Domains

- In a similar fashion, each row of the table can be kept as a list of the capabilities of that domain.
- Capability lists are associated with each domain, but not directly accessible by the domain or any user process.
- Capability lists are themselves protected resources, distinguished from other data in one of two ways:
    - A **tag,** possibly hardware implemented, distinguishing this special type of data. (other types may be floats, pointers, booleans, etc.)
    - The address space for a program may be split into multiple segments, at least one of which is inaccessible by the program itself, and used by the operating system for maintaining the process's access right capability list.

4.1.4 A Lock-Key Mechanism

- Each resource has a list of unique bit patterns, termed locks.

- Each domain has its own list of unique bit patterns, termed keys.
- Access is granted if one of the domain's keys fits one of the resource's locks.
- Again, a process is not allowed to modify its own keys.

4.1.5 Comparison

- Each of the methods here has certain advantages or disadvantages, depending on the particular situation and task at hand.
- Many systems employ some combination of the listed methods.

5. ACCESS CONTROL

- **Role-Based Access Control, RBAC,** assigns privileges to users, programs, or roles as appropriate, where "privileges" refer to the right to call certain system calls, or to use certain parameters with those calls.
- RBAC supports the principle of least privilege, and reduces the susceptibility to abuse as opposed to SUID or SGID programs.

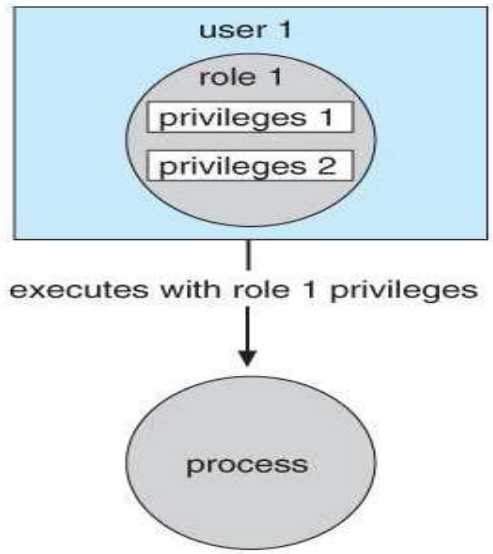

Figure - Role-based access control in Solaris 10.

6. Revocation of Access Rights

- The need to revoke access rights dynamically raises several questions:
  - Immediate versus delayed - If delayed, can we determine when the revocation will take place?
  - Selective versus general - Does revocation of an access right to an object affect *all* users who have that right, or only some users?
  - Partial versus total - Can a subset of rights for an object be revoked, or are all rights revoked at once?
  - Temporary versus permanent - If rights are revoked, is there a mechanism for processes to re-acquire some or all of the revoked rights?
- With an access list scheme revocation is easy, immediate, and can be selective, general, partial, total, temporary, or permanent, as desired.

- With capabilities lists the problem is more complicated, because access rights are distributed throughout the system. A few schemes that have been developed include:
    - Reacquisition - Capabilities are periodically revoked from each domain, which must then re-acquire them.
    - Back-pointers - A list of pointers is maintained from each object to each capability which is held for that object.
    - Indirection - Capabilities point to an entry in a global table rather than to the object. Access rights can be revoked by changing or invalidating the table entry, which may affect multiple processes, which must then re-acquire access rights to continue.
    - Keys - A unique bit pattern is associated with each capability when created, which can be neither inspected nor modified by the process.
        - A master key is associated with each object.
        - When a capability is created, its key is set to the object's master key.
        - As long as the capability's key matches the object's key, then the capabilities remain valid.
        - The object master key can be changed with the set-key command, thereby invalidating all current capabilities.
        - More flexibility can be added to this scheme by implementing a **list** of keys for each object, possibly in a global table.

Chapter-2

SECURITY

1 THE SECURITY PROBLEM

- Security deals with protecting systems from deliberate attacks, either internal or external, from individuals intentionally attempting to steal information, damage information, or otherwise deliberately wreak havoc in some manner.
- Some of the most common types of **violations** include:
    - ***Breach of Confidentiality*** **-** Theft of private or confidential information, such as credit-card numbers, trade secrets, patents, secret formulas, manufacturing procedures, medical information, financial information, etc.
    - ***Breach of Integrity*** **-** Unauthorized **modification** of data, which may have serious indirect consequences. For example a popular game or other program's source code could be modified to open up security holes on users systems before being released to the public.
    - ***Breach of Availability*** **-** Unauthorized **destruction** of data, often just for the "fun" of causing havoc and for bragging rites. Vandalism of web sites is a common form of this violation.
    - **Theft of Service -** Unauthorized use of resources, such as theft of CPU cycles, installation of daemons running an unauthorized file server, or tapping into the target's telephone or networking services.

- ***Denial of Service, DOS -*** Preventing legitimate users from using the system, often by overloading and overwhelming the system with an excess of requests for service.

- One common attack is **masquerading,** in which the attacker pretends to be a trusted third party. A variation of this is the **man-in-the-middle,** in which the attacker masquerades as both ends of the conversation to two targets.
- A **replay attack** involves repeating a valid transmission. Sometimes this can be the entire attack, or other times the content of the original message is replaced with malicious content.

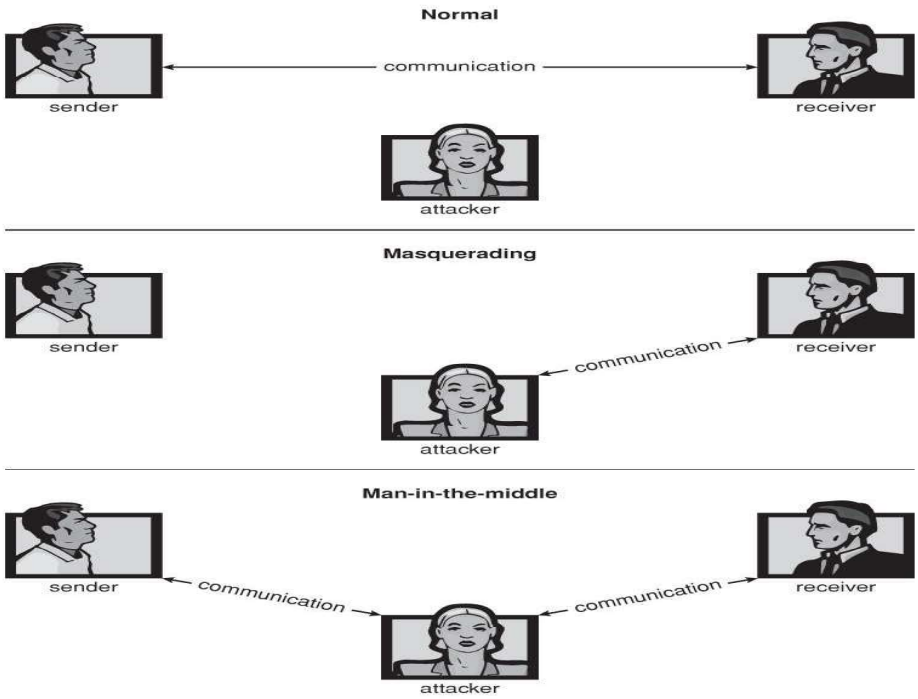

Figure 7.1 - Standard security attacks.

- There are four levels at which a system must be protected:
  1. **Physical -** The easiest way to steal data is to pocket the backup tapes. Also, access to the root console will often give the user special privileges, such as rebooting the system as root from removable media. Even general access to terminals in a computer room offers some opportunities for an attacker, although today's modern high-speed networking environment provides more and more opportunities for remote attacks.
  2. **Human -** There is some concern that the humans who are allowed access to a system be trustworthy, and that they cannot be coerced into breaching security. However more and more attacks today are made via **social engineering,** which basically means fooling trustworthy people into accidentally breaching security.
     - **Phishing** involves sending an innocent-looking e-mail or web site designed to fool people into revealing confidential information. E.g. spam e-mails pretending to be from e-Bay, PayPal, or any of a number of banks or credit-card companies.

    - **Dumpster Diving** involves searching the trash or other locations for passwords that are written down. (Note: Passwords that are too hard to remember, or which must be changed frequently are more likely to be written down somewhere close to the user's station.)
    - **Password Cracking** involves divining users passwords, either by watching them type in their passwords, knowing something about them like their pet's names, or simply trying all words in common dictionaries. (Note: "Good" passwords should involve a minimum number of characters, include non-alphabetical characters, and not appear in any dictionary (in any language), and should be changed frequently. Note also that it is proper etiquette to look away from the keyboard while someone else is entering their password.)
3. **Operating System -** The OS must protect itself from security breaches, such as runaway processes (denial of service), memory-access violations, stack overflow violations, the launching of programs with excessive privileges, and many others.
4. **Network -** As network communications become ever more important and pervasive in modern computing environments, it becomes ever more important to protect this area of the system. (Both protecting the network itself from attack, and protecting the local system from attacks coming in through the network.) This is a growing area of concern as wireless communications and portable devices become more and more prevalent.

2 PROGRAM THREATS

- There are many common threats to modern systems. Only a few are discussed here.

2.1 Trojan Horse

- A **Trojan Horse** is a program that secretly performs some maliciousness in addition to its visible actions.
- Some Trojan horses are deliberately written as such, and others are the result of legitimate programs that have become infected with **viruses,** (see below.)
- One dangerous opening for Trojan horses is long search paths, and in particular paths which include the current directory (".") as part of the path. If a dangerous program having the same name as a legitimate program (or a common mis-spelling, such as "sl" instead of "ls") is placed anywhere on the path, then an unsuspecting user may be fooled into running the wrong program by mistake.
- Another classic Trojan Horse is a login emulator, which records a users account name and password, issues a "password incorrect" message, and then logs off the system. The user then tries again (with a proper login prompt), logs in successfully, and doesn't realize that their information has been stolen.
- Two solutions to Trojan Horses are to have the system print usage statistics on logouts, and to require the typing of non-trappable key sequences such

as Control-Alt-Delete in order to log in. (This is why modern Windows systems require the Control-Alt-Delete sequence to commence logging in, which cannot be emulated or caught by ordinary programs. I.e. that key sequence always transfers control over to the operating system.)

- ***Spyware*** is a version of a Trojan Horse that is often included in "free" software downloaded off the Internet. Spyware programs generate pop-up browser windows, and may also accumulate information about the user and deliver it to some central site. (This is an example of **covert channels,** in which surreptitious communications occur.) Another common task of spyware is to send out spam e-mail messages, which then purportedly come from the infected user.

2.2 Trap Door

- A **Trap Door** is when a designer or a programmer (or hacker) deliberately inserts a security hole that they can use later to access the system.
- Because of the possibility of trap doors, once a system has been in an untrustworthy state, that system can never be trusted again. Even the backup tapes may contain a copy of some cleverly hidden back door.
- A clever trap door could be inserted into a compiler, so that any programs compiled with that compiler would contain a security hole. This is especially dangerous, because inspection of the code being compiled would not reveal any problems.

2.3 Logic Bomb

- A **Logic Bomb** is code that is not designed to cause havoc all the time, but only when a certain set of circumstances occurs, such as when a particular date or time is reached or some other noticeable event.
- A classic example is the **Dead-Man Switch**, which is designed to check whether a certain person (e.g. the author) is logging in every day, and if they don't log in for a long time (presumably because they've been fired), then the logic bomb goes off and either opens up security holes or causes other problems.

2.4 Stack and Buffer Overflow

- This is a classic method of attack, which exploits bugs in system code that allows buffers to overflow. Consider what happens in the following code, for example, if argv[1] exceeds 256 characters:
  - The strcpy command will overflow the buffer, overwriting adjacent areas of memory.
  - (The problem could be avoided using str**n**cpy, with a limit of 255 characters copied plus room for the null byte.)

```
#include
#define BUFFER_SIZE 256

int main( int argc, char * argv[ ] )
```

```
{
   char buffer[ BUFFER_SIZE ];

   if( argc < 2 )
      return -1;
   else {
      strcpy( buffer, argv[ 1 ] );
      return 0;
   }
}
```

Figure - C program with buffer-overflow condition.

- So how does overflowing the buffer cause a security breach? Well the first step is to understand the structure of the stack in memory:
  - The "bottom" of the stack is actually at a high memory address, and the stack grows towards lower addresses.
  - However the address of an array is the lowest address of the array, and higher array elements extend to higher addresses. (I.e. an array "grows" towards the bottom of the stack.
  - In particular, writing past the top of an array, as occurs when a buffer overflows with too much input data, can eventually overwrite the return address, effectively changing where the program jumps to when it returns.

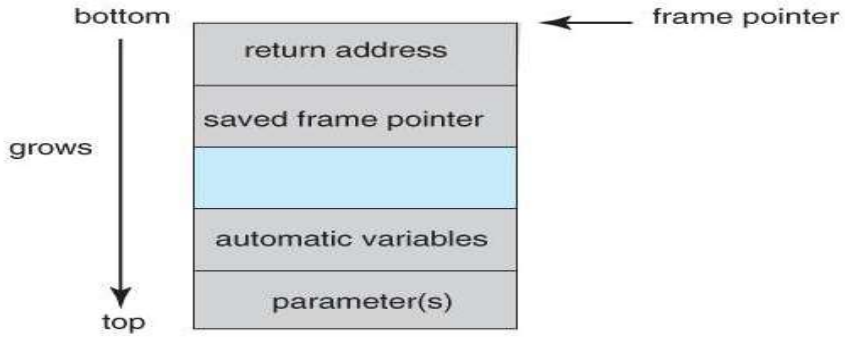

Figure - The layout for a typical stack frame.

- Now that we know how to change where the program returns to by overflowing the buffer, the second step is to insert some nefarious code, and then get the program to jump to our inserted code.
- Our only opportunity to enter code is via the input into the buffer, which means there isn't room for very much. One of the simplest and most obvious approaches is to insert the code for "exec(/bin/sh)". To do this requires compiling a program that contains this instruction, and then using an assembler or debugging tool to extract the minimum extent that includes the necessary instructions.
- The bad code is then padded with as many extra bytes as are needed to overflow the buffer to the correct extent, and the address of the buffer inserted into the return address location.

- The resulting block of information is provided as "input", copied into the buffer by the original program, and then the return statement causes control to jump to the location of the buffer and start executing the code to launch a shell.

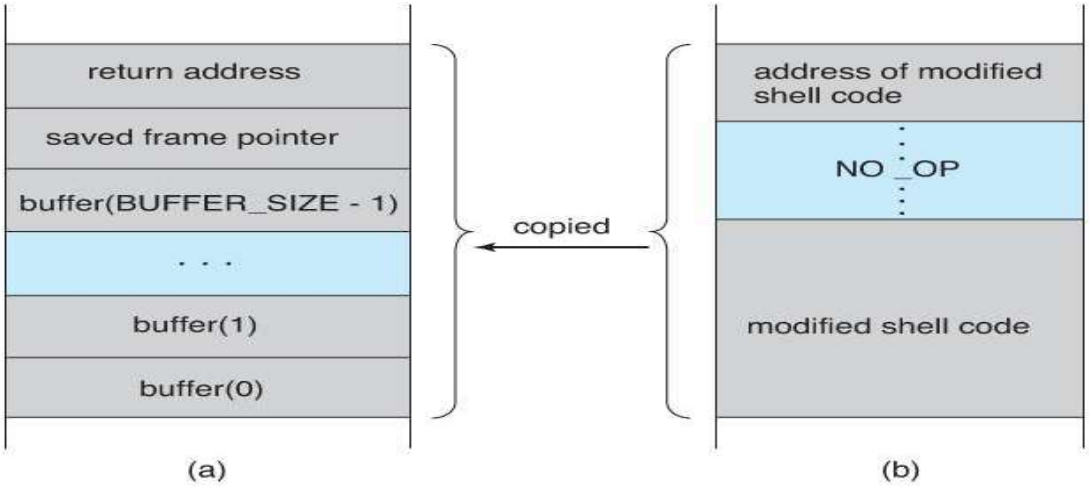

Figure - Hypothetical stack frame for Figure 7.2, (a) before and (b) after.

2.5 Viruses

- A virus is a fragment of code embedded in an otherwise legitimate program, designed to replicate itself (by infecting other programs), and (eventually) wreaking havoc.
- Viruses are more likely to infect PCs than UNIX or other multi-user systems, because programs in the latter systems have limited authority to modify other programs or to access critical system structures (such as the boot block.)
- Viruses are delivered to systems in a **virus dropper,** usually some form of a Trojan Horse, and usually via e-mail or unsafe downloads.
- Viruses take many forms (see below.) Figure 7.5 shows typical operation of a boot sector virus:

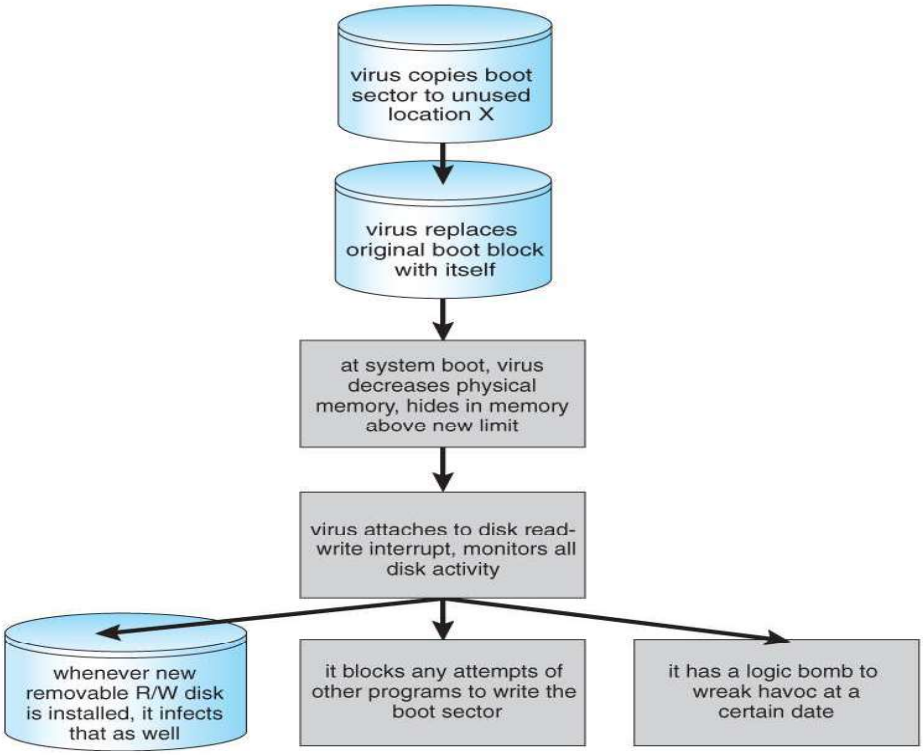

Figure - A boot-sector computer virus.

- Some of the forms of viruses include:
  - **File -** A file virus attaches itself to an executable file, causing it to run the virus code first and then jump to the start of the original program. These viruses are termed **parasitic,** because they do not leave any new files on the system, and the original program is still fully functional.
  - **Boot -** A boot virus occupies the boot sector, and runs before the OS is loaded. These are also known as **memory viruses**, because in operation they reside in memory, and do not appear in the file system.
  - **Macro -** These viruses exist as a macro (script) that are run automatically by certain macro-capable programs such as MS Word or Excel. These viruses can exist in word processing documents or spreadsheet files.
  - **Source code** viruses look for source code and infect it in order to spread.
  - **Polymorphic** viruses change every time they spread - Not their underlying functionality, but just their **signature,** by which virus checkers recognize them.
  - **Encrypted** viruses travel in encrypted form to escape detection. In practice they are self-decrypting, which then allows them to infect other files.
  - **Stealth** viruses try to avoid detection by modifying parts of the system that could be used to detect it. For example the read( ) system call could be modified so that if an infected file is read the infected part gets skipped and the reader would see the original unadulterated file.

- **Tunneling** viruses attempt to avoid detection by inserting themselves into the interrupt handler chain, or into device drivers.
- **Multipartite** viruses attack multiple parts of the system, such as files, boot sector, and memory.
- **Armored** viruses are coded to make them hard for anti-virus researchers to decode and understand. In addition many files associated with viruses are hidden, protected, or given innocuous looking names such as "...".

3 SYSTEM AND NETWORK THREATS

- Most of the threats described above are termed **program threats**, because they attack specific programs or are carried and distributed in programs. The threats in this section attack the operating system or the network itself, or leverage those systems to launch their attacks.

3.1 Worms

- A **worm** is a process that uses the fork / spawn process to make copies of itself in order to wreak havoc on a system. Worms consume system resources, often blocking out other, legitimate processes. Worms that propagate over networks can be especially problematic, as they can tie up vast amounts of network resources and bring down large-scale systems.
- One of the most well-known worms was launched by Robert Morris, a graduate student at Cornell, in November 1988. Targeting Sun and VAX computers running BSD UNIX version 4, the worm spanned the Internet in a matter of a few hours, and consumed enough resources to bring down many systems.
- This worm consisted of two parts:
  1. A small program called a **grappling hook,** which was deposited on the target system through one of three vulnerabilities, and
  2. The main worm program, which was transferred onto the target system and launched by the grappling hook program.

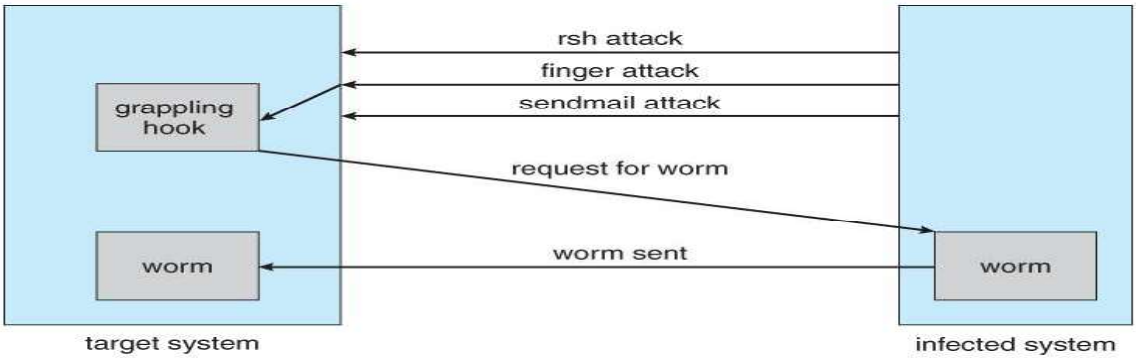

Figure- The Morris Internet worm.

- The three vulnerabilities exploited by the Morris Internet worm were as follows:

1. **rsh (remote shell)** is a utility that was in common use at that time for accessing remote systems without having to provide a password. If a user had an account on two different computers (with the same account name on both systems), then the system could be configured to allow that user to remotely connect from one system to the other without having to provide a password. Many systems were configured so that **any** user (except root) on system A could access the same account on system B without providing a password.
2. **finger** is a utility that allows one to remotely query a user database, to find the true name and other information for a given account name on a given system. For example "finger joeUser@somemachine.edu" would access the finger daemon at somemachine.edu and return information regarding joeUser. Unfortunately the finger daemon (which ran with system privileges) had the buffer overflow problem, so by sending a special 536-character user name the worm was able to fork a shell on the remote system running with root privileges.
3. **sendmail** is a routine for sending and forwarding mail that also included a debugging option for verifying and testing the system. The debug feature was convenient for administrators, and was often left turned on. The Morris worm exploited the debugger to mail and execute a copy of the grappling hook program on the remote system.

- Once in place, the worm undertook systematic attacks to discover user passwords:
    1. First it would check for accounts for which the account name and the password were the same, such as "guest", "guest".
    2. Then it would try an internal dictionary of 432 favorite password choices. (I'm sure "password", "pass", and blank passwords were all on the list.)
    3. Finally it would try every word in the standard UNIX on-line dictionary to try and break into user accounts.
- Once it had gotten access to one or more user accounts, then it would attempt to use those accounts to rsh to other systems, and continue the process.
- With each new access the worm would check for already running copies of itself, and 6 out of 7 times if it found one it would stop. (The seventh was to prevent the worm from being stopped by fake copies.)
- Fortunately the same rapid network connectivity that allowed the worm to propagate so quickly also quickly led to its demise - Within 24 hours remedies for stopping the worm propagated through the Internet from administrator to administrator, and the worm was quickly shut down.
- There is some debate about whether Mr. Morris's actions were a harmless prank or research project that got out of hand or a deliberate and malicious attack on the Internet. However the court system convicted him, and penalized him heavy fines and court costs.
- There have since been many other worm attacks, including the W32.Sobig.F@mm attack which infected hundreds of thousands of computers and an estimated 1 in 17 e-mails in August 2003. This worm made detection difficult by varying the subject line of the infection-carrying mail message, including "Thank You!", "Your details", and "Re: Approved".

3.2 Port Scanning

- **Port Scanning** is technically not an attack, but rather a search for vulnerabilities to attack. The basic idea is to systematically attempt to connect to every known (or common or possible) network port on some remote machine, and to attempt to make contact. Once it is determined that a particular computer is listening to a particular port, then the next step is to determine what daemon is listening, and whether or not it is a version containing a known security flaw that can be exploited.
- Because port scanning is easily detected and traced, it is usually launched from **zombie systems,** i.e. previously hacked systems that are being used without the knowledge or permission of their rightful owner. For this reason it is important to protect "innocuous" systems and accounts as well as those that contain sensitive information or special privileges.
- There are also port scanners available that administrators can use to check their own systems, which report any weaknesses found but which do not exploit the weaknesses or cause any problems. Two such systems are **nmap** (http://www.insecure.org/nmap) and **nessus** (http://www.nessus.org). The former identifies what OS is found, what firewalls are in place, and what services are listening to what ports. The latter also contains a database of known security holes, and identifies any that it finds.

3.3 Denial of Service

- **Denial of Service (DOS)** attacks do not attempt to actually access or damage systems, but merely to clog them up so badly that they cannot be used for any useful work. Tight loops that repeatedly request system services are an obvious form of this attack.
- DOS attacks can also involve social engineering, such as the Internet chain letters that say "send this immediately to 10 of your friends, and then go to a certain URL", which clogs up not only the Internet mail system but also the web server to which everyone is directed. (Note: Sending a "reply all" to such a message notifying everyone that it was just a hoax also clogs up the Internet mail service, just as effectively as if you had forwarded the thing.)
- Security systems that lock accounts after a certain number of failed login attempts are subject to DOS attacks which repeatedly attempt logins to all accounts with invalid passwords strictly in order to lock up all accounts.
- Sometimes DOS is not the result of deliberate maliciousness. Consider for example:
    - A web site that sees a huge volume of hits as a result of a successful advertising campaign.
    - CNN.com occasionally gets overwhelmed on big news days, such as Sept 11, 2001.
    - CS students given their first programming assignment involving fork( ) often quickly fill up process tables or otherwise completely consume system resources. :-)
    - (Please use ipcs and ipcrm when working on the inter-process communications assignment !)

4 CRYPTOGRAPHY AS A SECURITY TOOL

- Within a given computer the transmittal of messages is safe, reliable and secure, because the OS knows exactly where each one is coming from and where it is going.
- On a network, however, things aren't so straightforward - A rogue computer (or e-mail sender) may spoof their identity, and outgoing packets are delivered to a lot of other computers besides their (intended) final destination, which brings up two big questions of security:
    - **Trust -** How can the system be sure that the messages received are really from the source that they say they are, and can that source be trusted?
    - **Confidentiality -** How can one ensure that the messages one is sending are received only by the intended recipient?
- Cryptography can help with both of these problems, through a system of **secrets** and **keys.** In the former case, the key is held by the sender, so that the recipient knows that only the authentic author could have sent the message; In the latter, the key is held by the recipient, so that only the intended recipient can receive the message accurately.
- Keys are designed so that they cannot be divined from any public information, and must be guarded carefully. (**Asymmetric encryption** involve both a public and a private key.)

4.1 Encryption

- The basic idea of encryption is to encode a message so that only the desired recipient can decode and read it. Encryption has been around since before the days of Caesar, and is an entire field of study in itself. Only some of the more significant computer encryption schemes will be covered here.
- The basic process of encryption is shown in Figure 7.7, and will form the basis of most of our discussion on encryption. The steps in the procedure and some of the key terminology are as follows:
    1. The **sender** first creates a **message, m** in plaintext.
    2. The message is then entered into an **encryption algorithm, E,** along with the **encryption key, Ke.**
    3. The encryption algorithm generates the **ciphertext, c, = E(Ke)(m).** For any key k, E(k) is an algorithm for generating ciphertext from a message, and both E and E(k) should be efficiently computable functions.
    4. The ciphertext can then be sent over an unsecure network, where it may be received by **attackers.**
    5. The **recipient** enters the ciphertext into a **decryption algorithm, D,** along with the **decryption key, Kd.**
    6. The decryption algorithm re-generates the plaintext message, m, = D(Kd)(c). For any key k, D(k) is an algorithm for generating a clear text message from a ciphertext, and both D and D(k) should be efficiently computable functions.
    7. The algorithms described here must have this important property: Given a ciphertext c, a computer can only compute a message m such that c = E(k)(m) if it possesses D(k). (In other words, the

messages can't be decoded unless you have the decryption algorithm and the decryption key.)

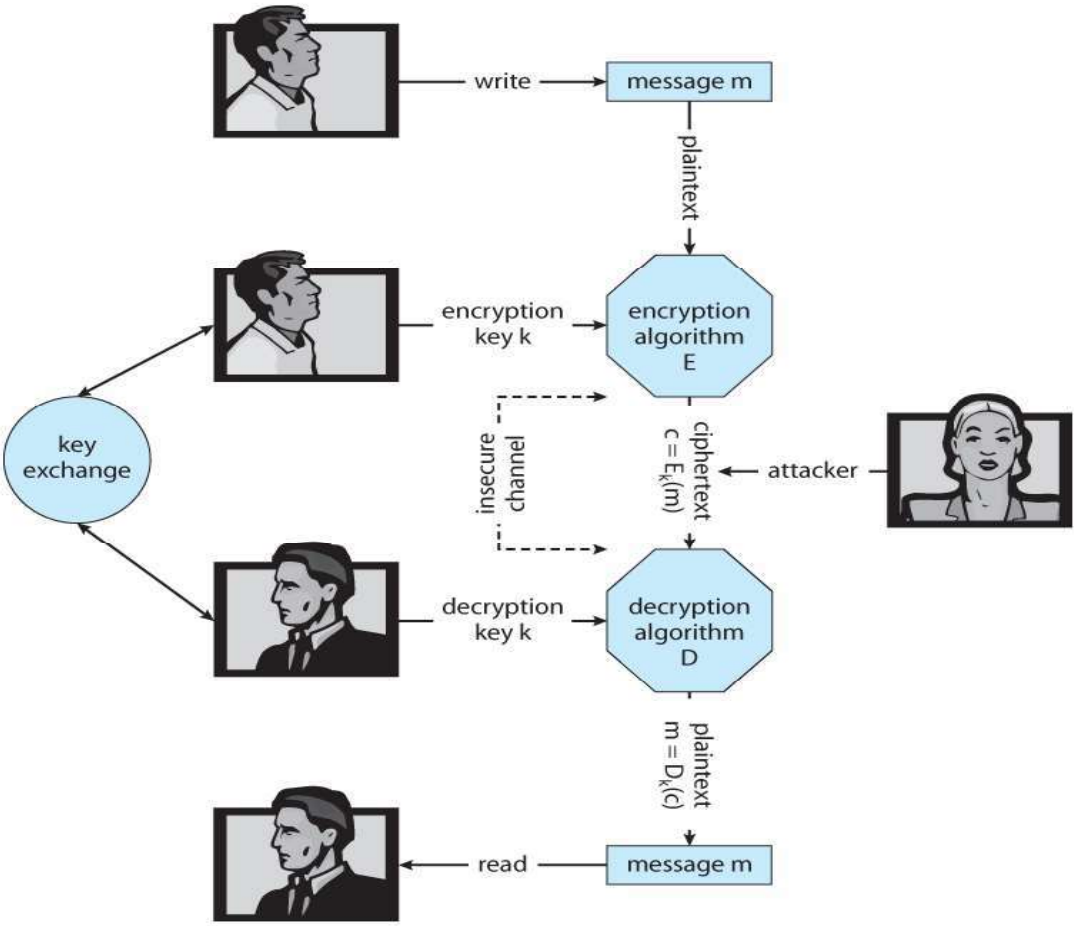

Figure 7.7 - A secure communication over an insecure medium.

4.1.1 Symmetric Encryption

- With **symmetric encryption** the same key is used for both encryption and decryption, and must be safely guarded. There are a number of well-known symmetric encryption algorithms that have been used for computer security:
  - The **Data-Encryption Standard, DES,** developed by the National Institute of Standards, NIST, has been a standard civilian encryption standard for over 20 years. Messages are broken down into 64-bit chunks, each of which are encrypted using a 56-bit key through a series of substitutions and transformations. Some of the transformations are hidden (black boxes), and are classified by the U.S. government.
  - DES is known as a **block cipher,** because it works on blocks of data at a time. Unfortunately this is a vulnerability if the same key is used for an extended amount of data. Therefore an enhancement is to not only encrypt each block, but also to XOR it with the previous block, in a technique known as **cipher-block chaining.**

- As modern computers become faster and faster, the security of DES has decreased, to where it is now considered insecure because its keys can be exhaustively searched within a reasonable amount of computer time. An enhancement called **triple DES** encrypts the data three times using three separate keys (actually two encryptions and one decryption) for an effective key length of 168 bits. Triple DES is in widespread use today.
- The **Advanced Encryption Standard, AES,** developed by NIST in 2001 to replace DES uses key lengths of 128, 192, or 256 bits, and encrypts in blocks of 128 bits using 10 to 6 rounds of transformations on a matrix formed from the block.
- The **twofish algorithm,** uses variable key lengths up to 256 bits and works on 128 bit blocks.
- ***RC5*** can vary in key length, block size, and the number of transformations, and runs on a wide variety of CPUs using only basic computations.
- ***RC4*** is a **stream cipher,** meaning it acts on a stream of data rather than blocks. The key is used to seed a pseudo-random number generator, which generates a **keystream** of keys. RC4 is used in **WEP,** but has been found to be breakable in a reasonable amount of computer time.

4.1.2 Asymmetric Encryption

- With **asymmetric encryption,** the decryption key, Kd, is not the same as the encryption key, Ke, and more importantly cannot be derived from it, which means the encryption key can be made publicly available, and only the decryption key needs to be kept secret. (or vice-versa, depending on the application.)
- One of the most widely used asymmetric encryption algorithms is **RSA,** named after its developers - Rivest, Shamir, and Adleman.
- RSA is based on two large prime numbers, ***p*** and **q,** (on the order of 512 bits each), and their product **N.**
  - Ke and Kd must satisfy the relationship: (Ke * Kd) % [(p - 1) * (q - 1)] = = 1
  - The encryption algorithm is: c = E(Ke)(m) = m^Ke % N
  - The decryption algorithm is: m = D(Kd)(c) = c^Kd % N
- An example using small numbers:
  - p = 7
  - q = 5
  - N = 7 * 5 = 91
  - (p - 1) * (q - 1) = 6 * 12 = 72
  - Select Ke < 72 and relatively prime to 72, say 5
  - Now select Kd, such that (Ke * Kd) % 72 = = 1, say 29
  - The public key is now (5, 91) and the private key is (29, 91)
  - Let the message, m = 42
  - Encrypt: c = 42^5 % 91 = 35
  - Decrypt: m = 35^29 % 91 = 42

- Note that asymmetric encryption is much more computationally expensive than symmetric encryption, and as such it is not normally used for large transmissions. Asymmetric encryption is suitable for small messages, authentication, and key distribution, as covered in the following sections.
-

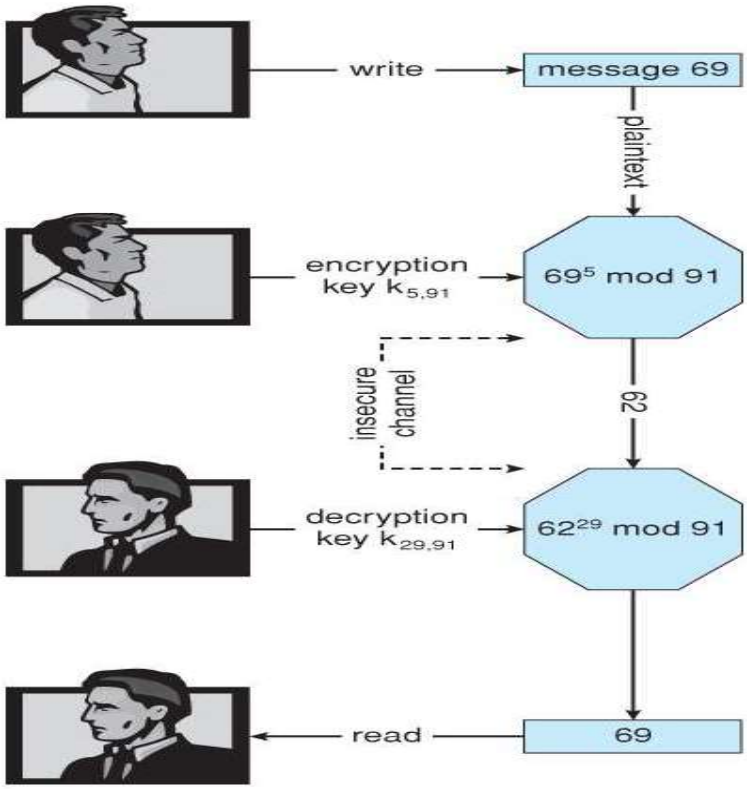

Figure - Encryption and decryption using RSA asymmetric cryptography

4.1.3 Authentication

- Authentication involves verifying the identity of the entity who transmitted a message.
- For example, if D(Kd)(c) produces a valid message, then we know the sender was in possession of E(Ke).
- This form of authentication can also be used to verify that a message has not been modified
- Authentication revolves around two functions, used for **signatures** (or **signing**), and **verification:**
    - A signing function, **S(Ks)** that produces an **authenticator, A,** from any given message m.
    - A Verification function, **V(Kv,m,A)** that produces a value of "true" if A was created from m, and "false" otherwise.
    - Obviously S and V must both be computationally efficient.
    - More importantly, it must not be possible to generate a valid authenticator, A, without having possession of S(Ks).
    - Furthermore, it must not be possible to divine S(Ks) from the combination of (m and A), since both are sent visibly across networks.

- Understanding authenticators begins with an understanding of hash functions, which is the first step:
    - **Hash functions, H(m)** generate a small fixed-size block of data known as a **message digest,** or **hash value** from any given input data.
    - For authentication purposes, the hash function must be **collision resistant on m.** That is it should not be reasonably possible to find an alternate message m' such that H(m') = H(m).
    - Popular hash functions are **MD5,** which generates a 128-bit message digest, and **SHA-1,** which generates a 160-bit digest.
- Message digests are useful for detecting (accidentally) changed messages, but are not useful as authenticators, because if the hash function is known, then someone could easily change the message and then generate a new hash value for the modified message. Therefore authenticators take things one step further by encrypting the message digest.
- A **message-authentication code, MAC,** uses symmetric encryption and decryption of the message digest, which means that anyone capable of verifying an incoming message could also generate a new message.
- An asymmetric approach is the **digital-signature algorithm,** which produces authenticators called **digital signatures.** In this case Ks and Kv are separate, Kv is the public key, and it is not practical to determine S(Ks) from public information. In practice the sender of a message signs it (produces a digital signature using S(Ks)), and the receiver uses V(Kv) to verify that it did indeed come from a trusted source, and that it has not been modified.
- There are three good reasons for having separate algorithms for encryption of messages and authentication of messages:
    1. Authentication algorithms typically require fewer calculations, making verification a faster operation than encryption.
    2. Authenticators are almost always smaller than the messages, improving space efficiency. (?)
    3. Sometimes we want authentication only, and not confidentiality, such as when a vendor issues a new software patch.

Another use of authentication is **non-repudiation,** in which a person filling out an electronic form cannot deny that they were the ones who did so.

4.1.4 Key Distribution

- Key distribution with symmetric cryptography is a major problem, because all keys must be kept secret, and they obviously can't be transmitted over unsecure channels. One option is to send them **out-of-band,** say via paper or a confidential conversation.
- Another problem with symmetric keys, is that a separate key must be maintained and used for each correspondent with whom one wishes to exchange confidential information.
- Asymmetric encryption solves some of these problems, because the public key can be freely transmitted through any channel, and the private key doesn't need to be transmitted anywhere. Recipients only need to maintain one private key for all incoming messages, though senders must maintain a separate public key for each recipient to which they might wish to send a

message. Fortunately the public keys are not confidential, so this **key-ring** can be easily stored and managed.

- Unfortunately there are still some security concerns regarding the public keys used in asymmetric encryption. Consider for example the following man-in-the-middle attack involving phony public keys:
- One solution to the above problem involves **digital certificates,** which are public keys that have been digitally signed by a trusted third party. But wait a minute - How do we trust that third party, and how do we know **they** are really who they say they are? Certain **certificate authorities** have their public keys included within web browsers and other certificate consumers before they are distributed. These certificate authorities can then vouch for other trusted entities and so on in a web of trust, as explained more fully in section 7.4.3.

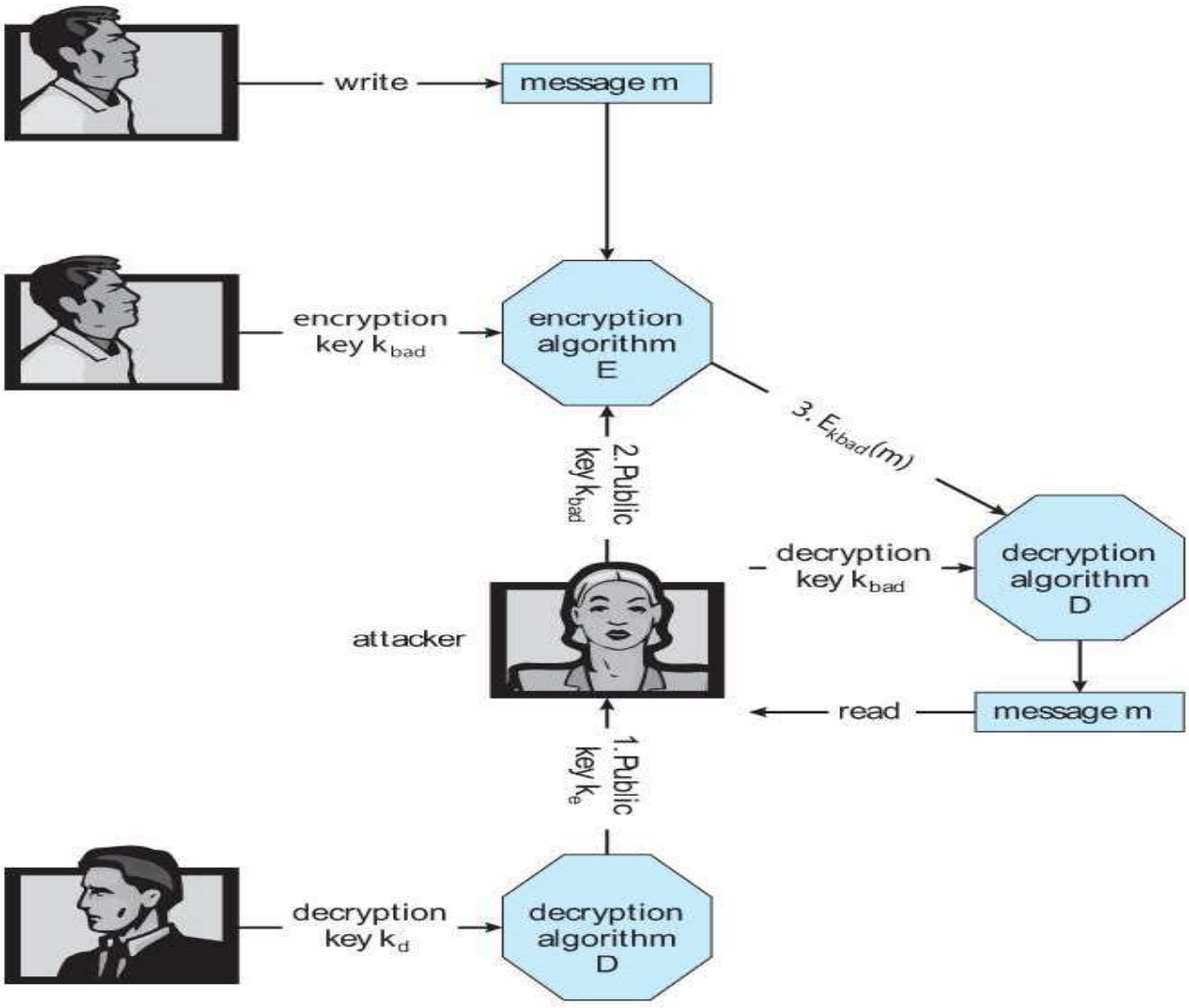

Figure - A man-in-the-middle attack on asymmetric cryptography.

4.2 Implementation of Cryptography

- Network communications are implemented in multiple layers - Physical, Data Link, Network, Transport, and Application being the most common breakdown.
- Encryption and security can be implemented at any layer in the stack, with pros and cons to each choice:
    - Because packets at lower levels contain the contents of higher layers, encryption at lower layers automatically encrypts higher layer information at the same time.
    - However security and authorization may be important to higher levels independent of the underlying transport mechanism or route taken.
- At the network layer the most common standard is **IPSec,** a secure form of the IP layer, which is used to set up **Virtual Private Networks, VPNs.**
- At the transport layer the most common implementation is SSL, described below.

4.3 An Example: SSL

- SSL (Secure Sockets Layer) 3.0 was first developed by Netscape, and has now evolved into the industry-standard TLS protocol. It is used by web browsers to communicate securely with web servers, making it perhaps the most widely used security protocol on the Internet today.
- SSL is quite complex with many variations, only a simple case of which is shown here.
- The heart of SSL is **session keys**, which are used once for symmetric encryption and then discarded, requiring the generation of new keys for each new session. The big challenge is how to safely create such keys while avoiding man-in-the-middle and replay attacks.
- Prior to commencing the transaction, the server obtains a **certificate** from a **certification authority, CA,** containing:
    - Server attributes such as unique and common names.
    - Identity of the public encryption algorithm, E( ), for the server.
    - The public key, k_e for the server.
    - The validity interval within which the certificate is valid.
    - A digital signature on the above issued by the CA:
        - a = S(K_CA)((attrs, E(k_e), interval)
- In addition, the client will have obtained a public **verification algorithm,** V(K_CA), for the certifying authority. Today's modern browsers include these built-in by the browser vendor for a number of trusted certificate authorities.
- The procedure for establishing secure communications is as follows:
    1. The client, c, connects to the server, s, and sends a random 28-byte number, n_c.
    2. The server replies with its own random value, n_s, along with its certificate of authority.
    3. The client uses its verification algorithm to confirm the identity of the sender, and if all checks out, then the client generates a 46 byte random **premaster secret, pms,** and sends an encrypted version of it as cpms = E(k_s)(pms)

4. The server recovers pms as D(k_s)(cpms).
5. Now both the client and the server can compute a shared 48-byte **master secret, ms,** = f(pms, n_s, n_c)
6. Next, both client and server generate the following from ms:
    - Symmetric encryption keys k_sc_crypt and k_cs_crypt for encrypting messages from the server to the client and vice-versa respectively.
    - MAC generation keys k_sc_mac and k_cs_mac for generating authenticators on messages from server to client and client to server respectively.
7. To send a message to the server, the client sends:
    - c = E(k_cs_crypt)(m, S(k_cs_mac))(m)))
8. Upon receiving c, the server recovers:
    - (m,a) = D(k_cs_crypt)(c)
    - and accepts it if V(k_sc_mac)(m,a) is true.

This approach enables both the server and client to verify the authenticity of every incoming message, and to ensure that outgoing messages are only readable by the process that originally participated in the key generation.
SSL is the basis of many secure protocols,including **Virtual Private Networks, VPNs,** in which private data is distributed over the insecure public internet structure in an encrypted fashion that emulates a privately owned network.

5 User Authentication

- A lot of chapter 6, Protection, dealt with making sure that only certain users were allowed to perform certain tasks, i.e. that a users privileges were dependent on his or her identity. But how does one verify that identity to begin with?

5.1 Passwords

- Passwords are the most common form of user authentication. If the user is in possession of the correct password, then they are considered to have identified themselves.
- In theory separate passwords could be implemented for separate activities, such as reading this file, writing that file, etc. In practice most systems use one password to confirm user identity, and then authorization is based upon that identification. This is a result of the classic trade-off between security and convenience.

5.2 Password Vulnerabilities

- Passwords can be guessed.
    - Intelligent guessing requires knowing something about the intended target in specific, or about people and commonly used passwords in general.

    - Brute-force guessing involves trying every word in the dictionary, or every valid combination of characters. For this reason good passwords should not be in any dictionary (in any language), should be reasonably lengthy, and should use the full range of allowable characters by including upper and lower case characters, numbers, and special symbols.
- "Shoulder surfing" involves looking over people's shoulders while they are typing in their password.
    - Even if the lurker does not get the entire password, they may get enough clues to narrow it down, especially if they watch on repeated occasions.
    - Common courtesy dictates that you look away from the keyboard while someone is typing their password.
    - Passwords echoed as stars or dots still give clues, because an observer can determine how many characters are in the password. :-(
- "Packet sniffing" involves putting a monitor on a network connection and reading data contained in those packets.
    - SSH encrypts all packets, reducing the effectiveness of packet sniffing.
    - However you should still never e-mail a password, particularly not with the word "password" in the same message or worse yet the subject header.
    - Beware of any system that transmits passwords in clear text. ("Thank you for signing up for XYZ. Your new account and password information are shown below".) You probably want to have a spare throw-away password to give these entities, instead of using the same high-security password that you use for banking or other confidential uses.
- Long hard to remember passwords are often written down, particularly if they are used seldomly or must be changed frequently. Hence a security trade-off of passwords that are easily divined versus those that get written down. :-(
- Passwords can be given away to friends or co-workers, destroying the integrity of the entire user-identification system.
- Most systems have configurable parameters controlling password generation and what constitutes acceptable passwords.
    - They may be user chosen or machine generated.
    - They may have minimum and/or maximum length requirements.
    - They may need to be changed with a given frequency. (In extreme cases for every session.)
    - A variable length history can prevent repeating passwords.
    - More or less stringent checks can be made against password dictionaries.

5.3 Encrypted Passwords

- Modern systems do not store passwords in clear-text form, and hence there is no mechanism to look up an existing password.

- Rather they are encrypted and stored in that form. When a user enters their password, that too is encrypted, and if the encrypted version match, then user authentication passes.
- The encryption scheme was once considered safe enough that the encrypted versions were stored in the publicly readable file "/etc/passwd".
    - They always encrypted to a 5 character string, so an account could be disabled by putting a string of any other length into the password field.
    - Modern computers can try every possible password combination in a reasonably short time, so now the encrypted passwords are stored in files that are only readable by the super user. Any password-related programs run as setuid root to get access to these files. (/etc/shadow)
    - A random seed is included as part of the password generation process, and stored as part of the encrypted password. This ensures that if two accounts have the same plain-text password that they will not have the same encrypted password. However cutting and pasting encrypted passwords from one account to another will give them the same plain-text passwords.

5.4 One-Time Passwords

- One-time passwords resist shoulder surfing and other attacks where an observer is able to capture a password typed in by a user.
    - These are often based on a **challenge** and a **response.** Because the challenge is different each time, the old response will not be valid for future challenges.
        - For example, The user may be in possession of a secret function f(x). The system challenges with some given value for x, and the user responds with f(x), which the system can then verify. Since the challenger gives a different (random) x each time, the answer is constantly changing.
        - A variation uses a map (e.g. a road map) as the key. Today's question might be "On what corner is SEO located?", and tomorrow's question might be "How far is it from Navy Pier to Wrigley Field?" Obviously "Taylor and Morgan" would not be accepted as a valid answer for the second question!
    - Another option is to have some sort of electronic card with a series of constantly changing numbers, based on the current time. The user enters the current number on the card, which will only be valid for a few seconds. A **two-factor authorization** also requires a traditional password in addition to the number on the card, so others may not use it if it were ever lost or stolen.
    - A third variation is a **code book,** or **one-time pad.** In this scheme a long list of passwords is generated, and each one is crossed off and cancelled as it is used. Obviously it is important to keep the pad secure.

5.5 Biometrics

- Biometrics involve a physical characteristic of the user that is not easily forged or duplicated and not likely to be identical between multiple users.
    - Fingerprint scanners are getting faster, more accurate, and more economical.
    - Palm readers can check thermal properties, finger length, etc.
    - Retinal scanners examine the back of the users' eyes.
    - Voiceprint analyzers distinguish particular voices.
    - Difficulties may arise in the event of colds, injuries, or other physiological changes.

6 IMPLEMENTING SECURITY DEFENSES

6.1 Security Policy

- A security policy should be well thought-out, agreed upon, and contained in a living document that everyone adheres to and is updated as needed.
- Examples of contents include how often port scans are run, password requirements, virus detectors, etc.

6.2 Vulnerability Assessment

- Periodically examine the system to detect vulnerabilities.
    - Port scanning.
    - Check for bad passwords.
    - Look for suid programs.
    - Unauthorized programs in system directories.
    - Incorrect permission bits set.
    - Program checksums / digital signatures which have changed.
    - Unexpected or hidden network daemons.
    - New entries in startup scripts, shutdown scripts, cron tables, or other system scripts or configuration files.
    - New unauthorized accounts.
- The government considers a system to be only as secure as its most far-reaching component. Any system connected to the Internet is inherently less secure than one that is in a sealed room with no external communications.
- Some administrators advocate "security through obscurity", aiming to keep as much information about their systems hidden as possible, and not announcing any security concerns they come across. Others announce security concerns from the rooftops, under the theory that the hackers are going to find out anyway, and the only one kept in the dark by obscurity are honest administrators who need to get the word.

6.3 Intrusion Detection

- Intrusion detection attempts to detect attacks, both successful and unsuccessful attempts. Different techniques vary along several axes:
    - The time that detection occurs, either during the attack or after the fact.

    - The types of information examined to detect the attack(s). Some attacks can only be detected by analyzing multiple sources of information.
    - The response to the attack, which may range from alerting an administrator to automatically stopping the attack (e.g. killing an offending process), to tracing back the attack in order to identify the attacker.
        - Another approach is to divert the attacker to a **honeypot**, on a **honeynet.** The idea behind a honeypot is a computer running normal services, but which no one uses to do any real work. Such a system should not see any network traffic under normal conditions, so any traffic going to or from such a system is by definition suspicious. Honeypots are normally kept on a honeynet protected by a **reverse firewall,** which will let potential attackers in to the honeypot, but will not allow any outgoing traffic. (So that if the honeypot is compromised, the attacker cannot use it as a base of operations for attacking other systems.) Honeypots are closely watched, and any suspicious activity carefully logged and investigated.
- Intrusion Detection Systems, IDSs, raise the alarm when they detect an intrusion. Intrusion Detection and Prevention Systems, IDPs, act as filtering routers, shutting down suspicious traffic when it is detected.
- There are two major approaches to detecting problems:
    - ***Signature-Based Detection*** scans network packets, system files, etc. looking for recognizable characteristics of known attacks, such as text strings for messages or the binary code for "exec /bin/sh". The problem with this is that it can only detect previously encountered problems for which the signature is known, requiring the frequent update of signature lists.
    - ***Anomaly Detection*** looks for "unusual" patterns of traffic or operation, such as unusually heavy load or an unusual number of logins late at night.
        - The benefit of this approach is that it can detect previously unknown attacks, so called **zero-day attacks.**
        - One problem with this method is characterizing what is "normal" for a given system. One approach is to benchmark the system, but if the attacker is already present when the benchmarks are made, then the "unusual" activity is recorded as "the norm."
        - Another problem is that not all changes in system performance are the result of security attacks. If the system is bogged down and really slow late on a Thursday night, does that mean that a hacker has gotten in and is using the system to send out SPAM, or does it simply mean that a CS 385 assignment is due on Friday? :-)
        - To be effective, anomaly detectors must have a very low **false alarm (false positive)** rate, lest the warnings get ignored, as well as a low **false negative** rate in which attacks are missed.

6.4 Virus Protection

- Modern anti-virus programs are basically signature-based detection systems, which also have the ability (in some cases) of **disinfecting** the affected files and returning them back to their original condition.
- Both viruses and anti-virus programs are rapidly evolving. For example viruses now commonly mutate every time they propagate, and so anti-virus programs look for families of related signatures rather than specific ones.
- Some antivirus programs look for anomalies, such as an executable program being opened for writing (other than by a compiler.)
- Avoiding bootleg, free, and shared software can help reduce the chance of catching a virus, but even shrink-wrapped official software has on occasion been infected by disgruntled factory workers.
- Some virus detectors will run suspicious programs in a **sandbox,** an isolated and secure area of the system which mimics the real system.
- ***Rich Text Format, RTF,*** files cannot carry macros, and hence cannot carry Word macro viruses.
- Known safe programs (e.g. right after a fresh install or after a thorough examination) can be digitally signed, and periodically the files can be re-verified against the stored digital signatures. (Which should be kept secure, such as on off-line write-only medium.)

6.5 Auditing, Accounting, and Logging

- Auditing, accounting, and logging records can also be used to detect anomalous behavior.
- Some of the kinds of things that can be logged include authentication failures and successes, logins, running of suid or sgid programs, network accesses, system calls, etc. In extreme cases almost every keystroke and electron that moves can be logged for future analysis. (Note that on the flip side, all this detailed logging can also be used to analyze system performance. The down side is that the logging also *affects* system performance (negatively!), and so a Heisenberg effect applies.)
- "The Cuckoo's Egg" tells the story of how Cliff Stoll detected one of the early UNIX break ins when he noticed anomalies in the accounting records on a computer system being used by physics researchers.

Tripwire Filesystem (New Sidebar)

- The tripwire filesystem monitors files and directories for changes, on the assumption that most intrusions eventually result in some sort of undesired or unexpected file changes.
- The tw.config file indicates what directories are to be monitored, as well as what properties of each file are to be recorded. (E.g. one may choose to monitor permission and content changes, but not worry about read access times.)
- When first run, the selected properties for all monitored files are recorded in a database. Hash codes are used to monitor file contents for changes.
- Subsequent runs report any changes to the recorded data, including hash code changes, and any newly created or missing files in the monitored directories.

- For full security it is necessary to also protect the tripwire system itself, most importantly the database of recorded file properties. This could be saved on some external or write-only location, but that makes it harder to change the database when legitimate changes are made.
- It is difficult to monitor files that are *supposed to* change, such as log files. The best tripwire can do in this case is to watch for anomalies, such as a log file that shrinks in size.
- Free and commercial versions are available at http://tripwire.org and http://tripwire.com.

7 FIREWALLING TO PROTECT SYSTEMS AND NETWORKS

- Firewalls are devices (or sometimes software) that sit on the border between two security domains and monitor/log activity between them, sometimes restricting the traffic that can pass between them based on certain criteria.
- For example a firewall router may allow HTTP: requests to pass through to a web server inside a company domain while not allowing telnet, ssh, or other traffic to pass through.
- A common architecture is to establish a de-militarized zone, DMZ, which sort of sits "between" the company domain and the outside world, as shown below. Company computers can reach either the DMZ or the outside world, but outside computers can only reach the DMZ. Perhaps most importantly, the DMZ cannot reach any of the other company computers, so even if the DMZ is breached, the attacker cannot get to the rest of the company network. (In some cases the DMZ may have limited access to company computers, such as a web server on the DMZ that needs to query a database on one of the other company computers.)

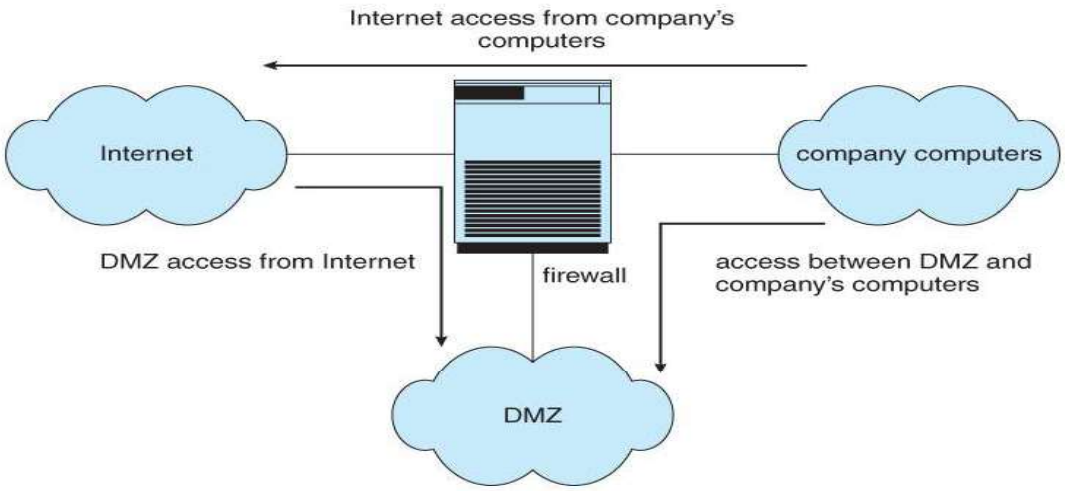

Figure - Domain separation via firewall.

- Firewalls themselves need to be resistant to attacks, and unfortunately have several vulnerabilities:
  - ***Tunneling,*** which involves encapsulating forbidden traffic inside of packets that are allowed.
  - Denial of service attacks addressed at the firewall itself.

    - Spoofing, in which an unauthorized host sends packets to the firewall with the return address of an authorized host.
- In addition to the common firewalls protecting a company internal network from the outside world, there are also some specialized forms of firewalls that have been recently developed:
    - A **personal firewall** is a software layer that protects an individual computer. It may be a part of the operating system or a separate software package.
    - An **application proxy firewall** understands the protocols of a particular service and acts as a stand-in (and relay) for the particular service. For example, and SMTP proxy firewall would accept SMTP requests from the outside world, examine them for security concerns, and forward only the "safe" ones on to the real SMTP server behind the firewall.
    - ***XML firewalls*** examine XML packets only, and reject ill-formed packets. Similar firewalls exist for other specific protocols.
    - ***System call firewalls*** guard the boundary between user mode and system mode, and reject any system calls that violate security policies.

8 Computer-Security Classifications

- No computer system can be 100% secure, and attempts to make it so can quickly make it unusable.
- However one can establish a level of trust to which one feels "safe" using a given computer system for particular security needs.
- The U.S. Department of Defense's "Trusted Computer System Evaluation Criteria" defines four broad levels of trust, and sub-levels in some cases:
    - Level D is the least trustworthy, and encompasses all systems that do not meet any of the more stringent criteria. DOS and Windows 3.1 fall into level D, which has no user identification or authorization, and anyone who sits down has full access and control over the machine.
    - Level C1 includes user identification and authorization, and some means of controlling what users are allowed to access what files. It is designed for use by a group of mostly cooperating users, and describes most common UNIX systems.
    - Level C2 adds individual-level control and monitoring. For example file access control can be allowed or denied on a per-individual basis, and the system administrator can monitor and log the activities of specific individuals. Another restriction is that when one user uses a system resource and then returns it back to the system, another user who uses the same resource later cannot read any of the information that the first user stored there. (I.e. buffers, etc. are wiped out between users, and are not left full of old contents.) Some special secure versions of UNIX have been certified for C2 security levels, such as SCO.
    - Level B adds sensitivity labels on each object in the system, such as "secret", "top secret", and "confidential". Individual users have different clearance levels, which controls which objects they are able

to access. All human-readable documents are labeled at both the top and bottom with the sensitivity level of the file.

- Level B2 extends sensitivity labels to all system resources, including devices. B2 also supports covert channels and the auditing of events that could exploit covert channels.
- B3 allows creation of access-control lists that denote users NOT given access to specific objects.
- Class A is the highest level of security. Architecturally it is the same as B3, but it is developed using formal methods which can be used to **prove** that the system meets all requirements and cannot have any possible bugs or other vulnerabilities. Systems in class A and higher may be developed by trusted personnel in secure facilities.
- These classifications determine what a system *can* implement, but it is up to security policy to determine *how* they are implemented in practice. These systems and policies can be reviewed and certified by trusted organizations, such as the National Computer Security Center. Other standards may dictate physical protections and other

10 MARKS QUESTIONS

1. (a) What is an Interrupt? Explain about Interrupt driven I/O Cycle?
   (b) What is DMA? Explain steps in DMA transfer operation?
2. Explain about importance and applications of I/O Interface?
3. Explain about services provided by Kernel I/O subsystem?
4. Explain the steps in transforming I/O request to Hardware operation with neat diagram?
5. Write short notes on
   (a) Principles of protection (b) Goals of Protection (c) Domain Structure
6. What it is Access Matrix? Explain different methods to implement Access Matrix?
7. (a) Write about Revocation of Access Rights?
   (b) Write a short note on Language based Protection?
8. Explain about different Program Threats?
9. Explain about different User Authentication mechanisms?
10. What is Encryption? Explain about Symmetric and Asymmetric Encryption mechanisms?