PRINCIPLES OF DEADLOCK

Syllabus

System Model
Deadlock Characterization
Deadlock Prevention
Detection and Avoidance
Recovery from Deadlock

System Model

- A system consists of a finite number of resources to be distributed among a number of competing processes.
- The resources are partitioned into several types, each consisting of some number of identical instances.
- Memory space, CPU cycles, files, and I/O devices (such as printers and DVD drives) are examples of resource types.
- If a system has two CPUs, then the resource type *CPU* has two instances. Similarly, the resource type *printer* may have five instances.
- If a process requests an instance of a resource type, the allocation of *any* instance of the type will satisfy the request.
- A process must request a resource before using it and must release the resource after using it.
- A process may request as many resources as it requires to carry out its designated task.
- Obviously, the number of resources requested may not exceed the total number of resources available in the system.
- In other words, a process cannot request three printers if the system has only two.
- Under the normal mode of operation, a process may utilize a resource in only the following sequence:
  1. **Request.** If the request cannot be granted immediately (for example, if the resource is being used by another process), then the requesting process must wait until it can acquire the resource.
  2. **Use,** The process can operate on the resource (for example, if the resource is a printer, the process can print on the printer).
  3. **Release.** The process releases the resource.
- The request and release of resources are system calls. Examples are the request () and release () device, open () and close () file, and allocate () and free () memory system calls.
- Request and release of resources that are not managed by the operating system can be accomplished through the wait () and signal () operations on semaphores or through acquisition and release of a mutex lock.
- A system table records whether each resource is free or allocated; for each resource that is allocated, the table also records the process to which it is allocated. If a process requests a

resource that is currently allocated to another process, it can be added to a queue of processes waiting for this resource.

- A set of processes is in a deadlock state when every process in the set is waiting for an event that can be caused only by another process in the set.
- The events with which we are mainly concerned here are resource acquisition and release.
- The resources maybe either physical resources (for example, printers, tape drives, memory space, and CPU cycles) or logical resources (for example, files, semaphores, and monitors). However, other types of events may result in deadlocks.
- To illustrate a deadlock state, consider a system with three CD RW drives.
- Suppose each of three processes holds one of these CD RW drives. If each process now requests another drive, the three processes will be in a deadlock state.
- Each is waiting for the event "CD RW is released," which can be caused only by one of the other waiting processes. This example illustrates a deadlock involving the same resource type.
- Deadlocks may also involve different resource types.
- For example, consider a system with one printer and one DVD drive. Suppose that process Pi is holding the DVD and process *Pj* is holding the printer. If Pi requests the printer and Pj requests the DVD drive, a deadlock occurs.

Deadlock characterization

In a deadlock, processes never finish executing, and system resources are tied up, preventing other jobs from starting.

Necessary Conditions

- A deadlock situation can arise if the following four conditions hold simultaneously in a system:
  1. **Mutual exclusion.** At least one resource must be held in a non-sharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

  2. **Hold and wait.** A process must be holding at least one resource and waiting to acquire additional resources that are currently being held by other processes.

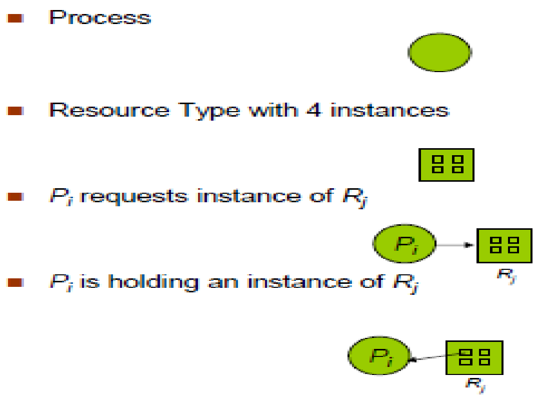
  3. **No preemption.** Resources cannot be preempted. That is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.

  4. **Circular wait.** A set {*P0, P1, ..., Pn}* of waiting processes must exist such that *P0* is waiting for a resource held by *P1, P1* is waiting for a resource held by P2, •••, Pn-1 is waiting for a resource held by *Pn,* and *Pn* is waiting for a resource held by P0.

We emphasize that all four conditions must hold for a deadlock to occur. The circular-wait condition implies the hold-and-wait condition, so the four conditions are not completely independent.

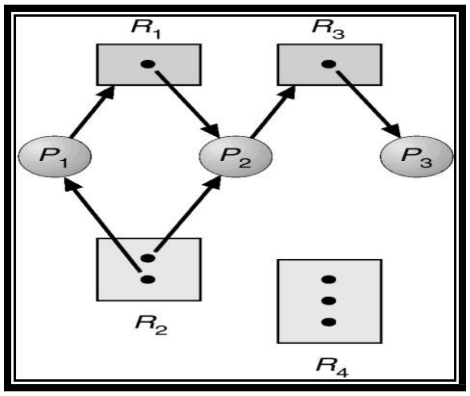
Resource-Allocation Graph

- Deadlocks can be described more precisely in terms of a directed graph called a **system resource-allocation** graph.
- This graph consists of a set of vertices *V* and a set of edges *E*. The set of vertices *V* is partitioned into two different types of nodes: P - {P1, *P2,.., Pn}* the set consisting of all the active processes in the system, and *R = {R1, R2, ••• Rm},* the set consisting of all resource types in the system.
- A directed edge from process *Pi* to resource type *Rj* is denoted by Pi -> Rj; it signifies that process Pi has requested an instance of resource type Rj and is currently waiting for that resource.
- A directed edge from resource type *Rj* to process *Pi* is denoted by *Rj* → Pi; it signifies that an instance of resource type *Rj* has been allocated to process Pi.
- A directed edge Pi—> *Rj* is called a **request edge;** a directed edge *Rj* → Pi is called an **assignment edge.**
- Pictorially, we represent each process Pi as a circle and each resource type *Rj* as a rectangle.
- Since resource type *Rj* may have more than one instance, we represent each such instance as a dot within the rectangle.
- Note that a request edge points to only the rectangle Rj, where as an assignment edge must also designate one of the dots in the rectangle.
- When process Pi requests an instance of resource type *Rj,* a request edge is inserted in the resource-allocation graph. When this request can be fulfilled, the request edge is *instantaneously* transformed to an assignment edge.
- When the process no longer needs access to the resource, it releases the resource; as a result, the assignment edge is deleted.


Process
Resource Type with 4 instances
P_i requests instance of R_j
P_i
R_j
P_i is holding an instance of R_j
P_i
R_j

- The sets P, *R*, and E:
  P={P1,P2,P3}
  R= {R1, R2, R3, R4}
  E = {P1→R1, *P2*→R3, R1→P2, *R2*→P2, *R2*→*P1*, *R3*→P3 }
- Resource instances:
  One instance of resource type R1
  Two instances of resource type R2
  One instance of resource type *R3*
  Three instances of resource type *R4*

- Process states:
    - Process *P1* is holding an instance of resource type R2 and is waiting for an instance of resource type R1.
    - Process P2 is holding an instance of *R1* and an instance of R2 and is waiting for an instance of *R3*.
    - Process P3 is holding an instance of R3.

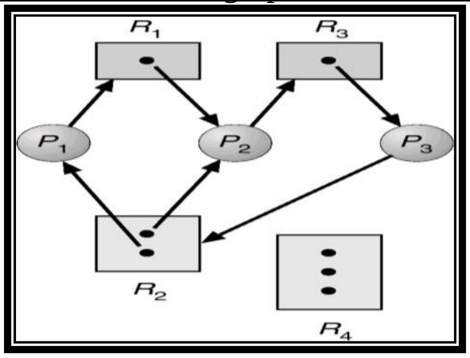


Resource-allocation graph.

- If the graph contains no cycles, then no process in the system is deadlocked.
- If the graph does contain a cycle, then a deadlock may exist.
- If each resource type has exactly one instance, then a cycle implies that a deadlock has occurred.
- If the cycle involves only a set of resource types, each of which has only a single instance, then a deadlock has occurred. Each process involved in the cycle is deadlocked.
- If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occurred.
- Suppose that process P3 requests an instance of resource type *R2*. Since no resource instance is currently available, a request edge P3 —>R2 is added to the graph .At this point, two minimal cycles exist in the system:

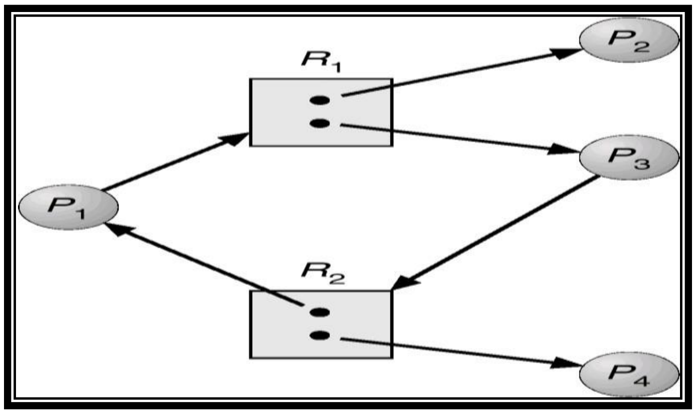
    P1 → R1 → P2 → R3 → P3 → R2 → P1
    P2 → R3 → P3 → R2 → P2

Resource-allocation graph with a deadlock.



Processes *P1, P2,* and P3 are deadlocked. Process P2 is waiting for the resource R3, which is held by process P3. Process P3 is waiting for either process *P1* or process *P2* to release resource R2. In addition, process P1 is waiting for process P2 to release resource R1.

Now consider the following resource-allocation graph.



Resource Allocation Graph With A Cycle But No Deadlock

In this example, we also have a cycle. However, there is no deadlock. Observe that process P4 may release its instance of resource type R_2. That resource can then be allocated to P_3, breaking the cycle.

In summary if a resource-allocation graph does not have a cycle, then the system is *not* in a deadlocked state. If there is a cycle, then the system may or may not be in a deadlocked state. This observation is important when we deal with the deadlock problem.

Methods for Handling Deadlocks

Generally speaking, we can deal with the deadlock problem in one of three ways:

- We can use a protocol to prevent or avoid deadlocks, ensuring that the system will *never* enter a deadlock state.
- We can allow the system to enter a deadlock state, detect it, and recover.
- We can ignore the problem altogether and pretend that deadlocks never occur in the system.

- ➢ The third solution is the one used by most operating systems, including UNIX and Windows; it is then up to **prevention** provides a set of methods for ensuring that at least one of the necessary conditions cannot hold. These methods prevent deadlocks by constraining how requests for resources can be made.
- ➢ **Deadlock avoidance** requires that the operating system be given in advance additional information concerning which resources a process will request and use during its lifetime. With this additional knowledge, it can decide for each request whether or not the process should wait. To decide whether the current request can be satisfied or must be delayed, the system must consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process.
- ➢ If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may arise.

- In this environment, the system can provide an algorithm that examines the state of the system to determine whether a deadlock has occurred and an algorithm to recover from the deadlock.
- If a the application developer to write programs that handle deadlocks.
- To ensure that deadlocks never occur, the system can use either a deadlock prevention or a deadlock-avoidance scheme.
- **Deadlock** system neither ensures that a deadlock will never occur nor provides a mechanism for deadlock detection and recovery, then we may arrive at a situation where the system is in a deadlocked state yet has no way of recognizing what has happened.

Deadlock Prevention

By ensuring that at least one of these conditions cannot hold, we can *prevent* the occurrence of a deadlock.

Mutual Exclusion

- The mutual-exclusion condition must hold for non-sharable resources.
- For example, a printer cannot be simultaneously shared by several processes.
- Sharable resources, in contrast, do not require mutually exclusive access and thus cannot be involved in a deadlock.
- Read-only files are a good example of a sharable resource.
- If several processes attempt to open a read-only file at the same time, they can be granted simultaneous access to the file.
- A process never needs to wait for a sharable resource.
- In general, however, we cannot prevent deadlocks by denying the mutual-exclusion condition, because some resources are intrinsically non-sharable,

Hold and Wait

- To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that, whenever a process requests a resource, it does not hold any other resources.
- One protocol that can be used requires each process to request and be allocated all its resources before it begins execution.
- We can implement this provision by requiring that system calls requesting resources for a process precede all other system calls.
- An alternative protocol allows a process to request resources only when it has none.
- A process may request some resources and use them. Before it can request any additional resources, however, it must release all the resources that it is currently allocated.
- To illustrate the difference between these two protocols, we consider a process that copies data from a DVD drive to a file on disk and then prints the results to a printer.
- If all resources must be requested at the beginning of the process, then the process must initially request the DVD drive, disk file, and printer.
- It will hold the printer for its entire execution, even though it needs the printer only at the end.
- The second method allows the process to request initially only the DVD drive and disk file.

- It copies from the DVD drive to the disk and then releases both the DVD drive and the disk file.
- The process must then again request the disk file and the printer. After copying the disk file to the printer, it releases these two resources and terminates.
- Both these protocols have two main disadvantages. First, resource utilization may be low, since resources may be allocated but unused for a long period.
- In the example given, for instance, we can release the DVD drive and disk file, and then again request the disk file and printer, only if we can be sure that our data will remain on the disk file.
- If we cannot be assured that they will, then we must request all resources at the beginning for both protocols.
- Second, starvation is possible. A process that needs several popular resources may have to wait indefinitely, because at least one of the resources that it needs is always allocated to some other process.
- The third necessary condition for deadlocks is that there be no preemption of resources that have already been allocated.
- To ensure that this condition does not hold, we can use the following protocol. If a process is holding some resources and requests another resource that cannot be immediately allocated to it , then all resources currently being held are preempted.
- In other words, these resources are implicitly released.
- The preempted resources are added to the list of resources for which the process is waiting.
- The process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.
- Alternatively, if a process requests some resources, we first check whether they are available. If they are, we allocate them.
- If they are not, we check whether they are allocated to some other process that is waiting for additional resources. If so, we preempt the desired resources from the waiting process and allocate them to the requesting process.
- If the resources are neither available nor held by a waiting process, the requesting process must wait.
- While it is waiting, some of its resources may be preempted, but only if another process requests them.
- A process can be restarted only when it is allocated the new resources it is requesting and recovers any resources that were preempted while it was waiting.
- This protocol is often applied to resources whose state can be easily saved and restored later, such as CPU registers and memory space. It cannot generally be applied to such resources as printers and tape drives.

No Preemption

- The third necessary condition for deadlocks is that there be no preemption of resources that have already been allocated.
- To ensure that this condition does not hold, we can use the following protocol.

- If a process is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the process must wait), then all resources currently being held are preempted.
- In other words, these resources are implicitly released. The preempted resources are added to the list of resources for which the process is waiting.
- The process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.
- Alternatively, if a process requests some resources, we first check whether they are available. If they are, we allocate them.
- If they are not, we check whether they are allocated to some other process that is waiting for additional resources. If so, we preempt the desired resources from the waiting process and allocate them to the requesting process.
- If the resources are neither available nor held by a waiting process, the requesting process must wait.
- While it is waiting, some of its resources may be preempted, but only if another process requests them.
- A process can be restarted only when it is allocated the new resources it is requesting and recovers any resources that were preempted while it was waiting.
- This protocol is often applied to resources whose state can be easily saved and restored later, such as CPU registers and memory space. It cannot generally be applied to such resources as printers and tape drives.

Circular Wait

- One way to ensure that this condition never holds is to impose a total ordering of all resource types and to require that each process requests resources in an increasing order of enumeration.
- To illustrate, we let $R = \{R1, R2, ..., Rm\}$ be the set of resource types. We assign to each resource type a unique integer number, which, allows us to compare two resources and to determine whether one precedes another in our ordering.
- Formally, we define a one-to-one function F: $R \longrightarrow N$, where N is the set of natural numbers. For example, if the set of resource types R includes tape drives, disk drives, and printers, then the function F might be defined as follows:

  F(tape drive) = 1
  F(disk drive) = 5
  F (printer) = 12
- We can now consider the following protocol to prevent deadlocks:
- Each process can request resources only in an increasing order of enumeration.
- That is, a process can initially request any number of instances of a resource type- say, Ri. After that, the process can request instances of resource type Rj if and only if F(Rj) > F(Ri).
- If several instances of the same resource type are needed, a *single* request for all of them must be issued.
- For example, using the function defined previously, a process that wants to use the tape drive and printer at the same time must first request the tape drive and then request the printer.

- Alternatively, we can require that, whenever a process requests an instance of resource type Rj, it has released any resources *Ri* such that *F(Ri) >= F(Rj).*
- If these two protocols are used, then the circular-wait condition cannot hold.
- We can demonstrate this fact by assuming that a circular wait exists.
- Let the set of processes involved in the circular wait be {P0, *P1,...,* Pn,}, where *Pi* is waiting for a resource Ri, which is held by process Pi+1.
- Then, since process Pi+1 is holding resource Ri, while requesting resource Ri+1, we must have F(Ri) < F(Ri+1), for all *i.* But this condition means that F(R()) < *F(R1)* < ••• < F(Rn) < *F(R0).* By transitivity, F(Ro) < *F(RQ),* which is impossible.
- Therefore, there can be no circular wait.

6.5 Deadlock Avoidance

Deadlock-prevention algorithms prevent deadlocks by restraining how requests can be made.

- The restraints ensure that at least one of the necessary conditions for deadlock cannot occur and, hence, that deadlocks cannot hold.
- Possible side effects of preventing deadlocks by this method, however, are low device utilization and reduced system throughput.
- An alternative method for avoiding deadlocks is to require additional information about how resources are to be requested.
- For example, in a system with one tape drive and one printer, the system might need to know that process P will request first the tape drive and then the printer before releasing both resources, whereas process *Q* will request first the printer and then the tape drive.
- With this knowledge of the complete sequence of requests and releases for each process, the system can decide for each request whether or not the process should wait in order to avoid a possible future deadlock.
- Each request requires that in making this decision the system consider the resources currently available, the resources currently allocated to each process, and the future requests and releases of each process.
- The various algorithms that use this approach differ in the amount and type of information required.
- The simplest and most useful model requires that each process declare the *maximum number* of resources of each type that it may need.
- Given this a priori, information, it is possible to construct an algorithm that ensures that the system will never enter a deadlocked state.
- Such an algorithm defines the deadlock-avoidance approach.
- A deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular wait condition can never exist.
- The resource-allocation *state* is defined by the number of available and allocated resources and the maximum demands of the processes. In the following sections, we explore two deadlock-avoidance algorithms.

Safe State

- A state is *safe* if the system can allocate resources to each process in some order and still avoid a deadlock.
- More formally, a system is in a safe state only if there exists a safe sequence.
- Sequence <*P*1, *P*2, …, *Pn*> is safe if for each *P*i, the resources that *Pi* can still request can be satisfied by currently available resources + resources held by all the *Pj*, with *j<I*.
  - If Pi resource needs are not immediately available, then *Pi* can wait until all *Pj* have finished.
  - When *Pj* is finished, *P*i can obtain needed resources, execute, return allocated resources, and terminate.
  - When *Pi* terminates, *Pi*+1 can obtain its needed resources, and so on.
- If a system is in safe state ⇒ no deadlocks.
- If a system is in unsafe state ⇒ possibility of deadlock.
- Avoidance ⇒ ensure that a system will never enter an unsafe state.

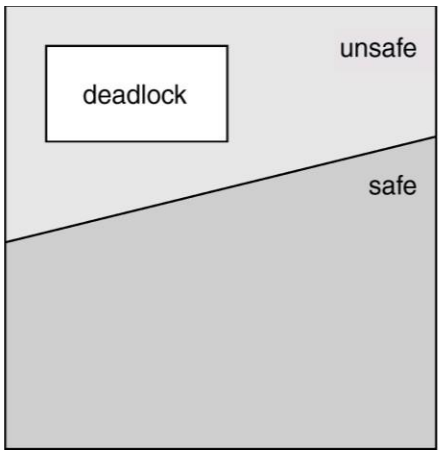


Fig: Safe, unsafe, and deadlock state spaces.

- To illustrate, we consider a system with 12 magnetic tape drives and three processes: P0, *P1*, and P2. Process P0 requires 10 tape drives, process P1 may need as many as 4 tape drives, and process P2 may need up to 9 tape drives.
- Suppose that, at time *to,* process P0 is holding 5 tape drives, process *P1* is holding *2* tape drives, and process *P2* is holding 2 tape drives.

Process	Allocated	Maximum Needs	Current Needs	Available
P0	5	10	5	3
P1	2	4	2	
P2	2	9	7	

- At time to, the system is in a safe state. The sequence < P1, *Po, P2*> satisfies the safety condition.
- Process P1 can immediately be allocated all its tape drives and then return them (the system will then have 5 available tape drives); then process P0 can get all its tape drives and return

them (the system will then have 10 available tape drives); and finally process P2 can get all its tape drives and return them (the system will then have all 12 tape drives available).

- A system can go from a safe state to an unsafe state. Suppose that, at time *t1,* process *P2* requests and is allocated one more tape drive.

Process	Allocated	Maximum Needs	Current Needs	Available
P0	5	10	5	2
P1	2	4	2	
P2	3	9	6	

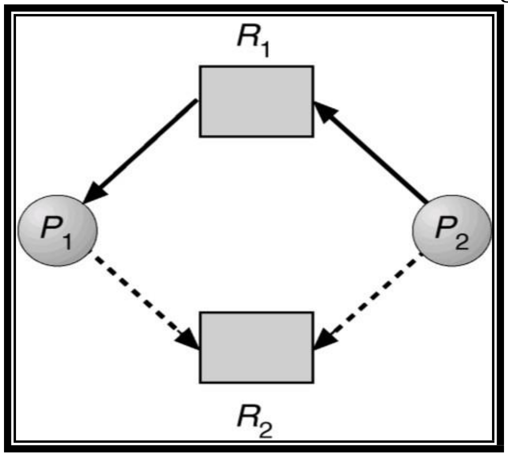
- The system is no longer in a safe state. At this point, only process P1, can be allocated all its tape drives. When it returns them, the system will have only 4 available tape drives.
- Since process *P0,* is allocated 5 tape drives but has a maximum of 10, it may request 5 more tape drives. Since they are unavailable, process Po must wait.
- Similarly, process P2 may request an additional 6 tape drives and have to wait, resulting in a deadlock.
- Our mistake was in granting the request from process *P2* for one more tape drive. If we had made P2 wait until either of the other processes had finished and released its resources, then we could have avoided the deadlock.

Resource-Allocation-Graph Algorithm

- If we have a resource-allocation system with only one instance of each resource type, we use a variant of the resource-allocation graph algorithm for deadlock avoidance.
- In addition to the request and assignment edges already described, we introduce a new type of edge, called a claim edge.
- A claim edge Pi—> *Rj* indicates that process *Pi* may request resource *Rj* at some time in the future. This edge resembles a request edge in direction but is represented in the graph by a dashed line.
- When process Pi requests resource *Rj*, the claim edge Pi —> *Rj* is converted to a request edge.
- Similarly, when a resource *Rj* is released by Pi, the assignment edge *Rj* → Pi is reconverted to a claim edge *Pi —> Rj.*
- We note that the resources must be claimed a priori in the system. That is, before process Pi starts executing, all its claim edges must already appear in the resource-allocation graph.
- Suppose that process Pi requests resource *Rj.* The request can be granted only if converting the request edge Pi —» *Rj* to an assignment edge *Rj* —> *Pi* does not result in the formation of a cycle in the resource-allocation graph.
- Note that we check for safety by using a cycle-detection algorithm. An algorithm for detecting a cycle in this graph requires an order of n^2 operations, where *n* is the number of processes in the system.
- If no cycle exists, then the allocation of the resource will leave the system in a safe state.
- If a cycle is found, then the allocation will put the system in an unsafe state.

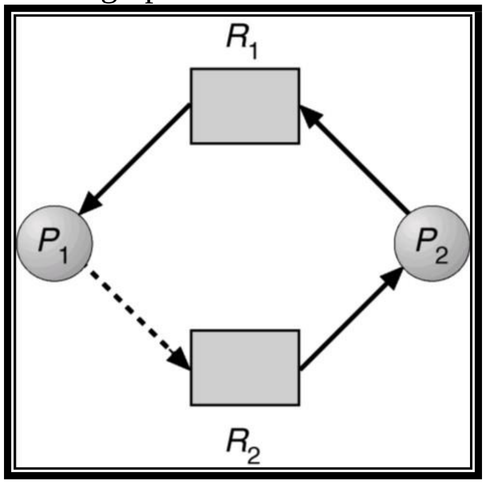
- Therefore, process *Pi* will have to wait for its requests to be satisfied.

- To illustrate this algorithm, we consider the resource-allocation graph.



Resource-allocation graph for deadlock avoidance.

- Suppose that *P2* requests *R2.* Although *R2* is currently free, we cannot allocate it toP2, since this action will create a cycle in the graph as shown in below .



An unsafe state in a resource-allocation graph.

- A cycle indicates that the system is in an unsafe state. If P1 requests *R2,* and *P2* requests *R1* then a deadlock will occur.

Banker's Algorithm

- The resource-allocation-graph algorithm is not applicable to a resource allocation system with multiple instances of each resource type.
- The deadlock avoidance algorithm that we describe next is applicable to such a system but is less efficient than the resource-allocation graph scheme.

- This algorithm is commonly known as the *banker's algorithm.* The name was chosen because the algorithm could be used in a banking system.
- When, a new process enters the system, it must declare the maximum number of instances of each resource type that it may need.
- This number may not exceed the total number of resources in the system.
- When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state.
- If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources.
- Several data structures must be maintained to implement the banker's algorithm.
- These data structures encode the state of the resource-allocation system.
- Let *n* be the number of processes in the system and *m* be the number of resource types.
- We need the following data structures:
  - **Available**. A vector of length *m* indicates the number of available resources of each type. If *Available[j]* equals *k,* there are *k* instances of resource type *Rj* available.
  - **Max.** An *n* x *m* matrix defines the maximum demand of each process. If Max[i][j] equals *k,* then process *Pi* may request at most *k* instances of resource type Rj.
  - **Allocation.** An *n* x *in* matrix defines the number of resources of each type currently allocated to each process. If *Allocation[i][j]* equals *k,* then process *Pi* is currently allocated *k* instances of resource type Rj.
  - **Need**. An *n* x *m* matrix indicates the remaining resource need of each process. If *Need[i][j]* equals *k,* then process Pi may need *k* more instances of resource type *Rj* to complete its task. Note that *Need*[i][j] equals *Max[i][j] - Allocation[i][j].*

- These data structures vary over time in both size and value.
- We can treat each row in the matrices *Allocation* and *Need* as vectors and refer to them as $Allocation_i$ and $Need_i$.
- The vector $Allocation_i$ specifies the resources currently allocated to process Pi; the vector $Need_i$ specifies the additional resources that process Pi may still request to complete its task.

Safety Algorithm

- We can now present the algorithm for finding out whether or not a system is in a safe state.
- This algorithm can be described, as follows:

```
1. Let Work and Finish be vectors of length m and n, respectively. Initialize Work = Available
   and Finish[i] = false for i= 0 , 1 , ..., n -1 .
2. Find an i such that both
   a. Finish[i] ==false
   b. $Need_i < Work$
3. If no such i exists, go to step 4.
4. $Work = Work + Allocation_i$
   Finish[i] = true
   Go to step 2.
```

5. If *Finish[i] = true* for all. i , then the system is in a safe state.

- This algorithm may require an order of *m* x n^2 operations to determine whether a state is safe.

Resource-Request Algorithm

- We now describe the algorithm which determines if requests can be safely granted.
- Let *Request*$_i$ be the request vector for process Pi. If *Request*$_i$ *[j] = k*, then process *Pi* wants *k* instances of resource type *Rj*.
- When a request for resources is made by process *Pi,* the following actions are taken:

1. If *Request*$_i$ < *Need*$_i$, go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If *Request*$_i$ < *Available*$_i$ go to step 3. Otherwise, *Pi* must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to process Pi by modifying the state as follows:
   Available = Available - Request$_i$
   Allocation$_i$*, = Allocation*$_i$ *+ Request*$_i$
   Need$_i$ *= Need*$_i$ *- Request*$_i$

- If the resulting resource-allocation state is safe, the transaction is completed, and process Pi is allocated its resources.
- However, if the new state is unsafe, then *Pi* must wait for *Request*$_i$,and the old resource-allocation state is restored.

An Illustrative Example

- Finally, to illustrate the use of the banker's algorithm, consider a system with five processes *P0* through P4 and three resource types *A, B,* and C.
- Resource type A has 10 instances, resource type *B* has 5 instances, and resource type *C* has 7 instances.
- Suppose that, at time To, the following snapshot of the system has been taken:

	Allocation	Max	Available
	A B C	A B C	A B C
P0	0 1 0	7 5 3	3 3 2
P1	2 0 0	3 2 2	
P2	3 0 2	9 0 2	
P3	2 1 1	2 2 2	
P4	0 0 2	4 3 3	

- The content of the matrix *Need* is defined to be *Max - Allocation* and is as follows:

Need
A B C

P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

- Currently the system is in safe state.

Safe sequence: Safe sequence is calculated as follows:

1) Need of each process is compared with available. If need. < available;, then the resources are allocated to that process and process will release the resource.
2) If need is greater than available, next process need is taken for comparison.
3) In the above example, need of process P. is (7, 4, 3) and available is (3, 3, 2).
   need > available —> False So system will move for next process.
4) Need for process P2 is (1, 2, 2) and available (3, 3, 2), so
   need < available (work) (1, 2, 2) < (3, 3, 2) = True then Finish [i] = True
   Request of P, is granted and processes P_2 is release the resource to the system.
   Work : = Work + Allocation Work : = (3, 3, 2) + (2, 0, 0) : = (5, 3, 2)
   This procedure is continued for all processes.
5) Next process P_3 need (6, 0,0) is compared with new available (5, 3,2).
   Need > Available = False (6 0 0) > (5 3 2)
6) Process P_4 need (0, 1, 1) is compared with available (5, 3, 2).
   Need < available
   (0 1 1) < (5 3 2) = True
   Available = Available + Allocation
   = (5 3 2) + (2 1 1) = (7 4 3) (New available)
7) Then process P_5 need (4 3 1) is compared with available (7 4 3)
   Need < Available
   (4 3 1) < (7 4 3) = True
   Available = Available + Allocation
   = (7 4 3) + (0 0 2) = (7 4 5) (New available)
8) Process P_1need (7 4 3) and available (7 4 5). If this request is granted then the system may be in the deadlock state. After granting the request, available resource is (0 0 2) so the system is in unsafe state.
9) Process P_3 need is (6 0 0) and available (7 4 5)

.'. Need < Available (6 0 0) < (7 4 5) = True Available = Available + Allocation
= (7 4 5) + (3 0 2) = (10 4 7) = (New available)

10) Last the remaining process P_1 need (7 4 3) and available is (10 4 7)
   Need < Available (7 4 3) < (10 4 7) = True

Available = Available + Allocation = (10 4 7) + (0 1 0) = (10 5 7)

Safe sequence is < $P_1P_4P_5P_0$ P_2>

Weakness of Banker's algorithm:

1. It requires that there be a fixed number of resources to allocate.
2. The algorithm requires that users sLate their maximum needs (request) in advance.
3. Number of users must remain fixed.
4. The algorithm requires that the bankers grant all requests within a finite time.
5. Algorithm requires that process returns ail resource within a finite time.

- Indeed, the sequence <*P1,* P3, *P4, P2, P0*> satisfies the safety criteria.
- Suppose now that process *P1* requests one additional instance of resource type *A* and two instances of resource type C, so $Request_1$= (1,0,2). To decide whether this request can be immediately granted, we first check that $Request_1$ < *Available*—that is, that (1,0,2) < (3,3,2), which is true. We then pretend that this request has been fulfilled, and we arrive at the following new state:

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 1	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

- We must determine whether this new system state is safe. To do so, we execute our safety algorithm and find that the sequence <*P1, P3, P4,* P0, *P2*> satisfies the safety requirement.
- Hence, we can immediately grant the request of process *P1.*

Deadlock Detection

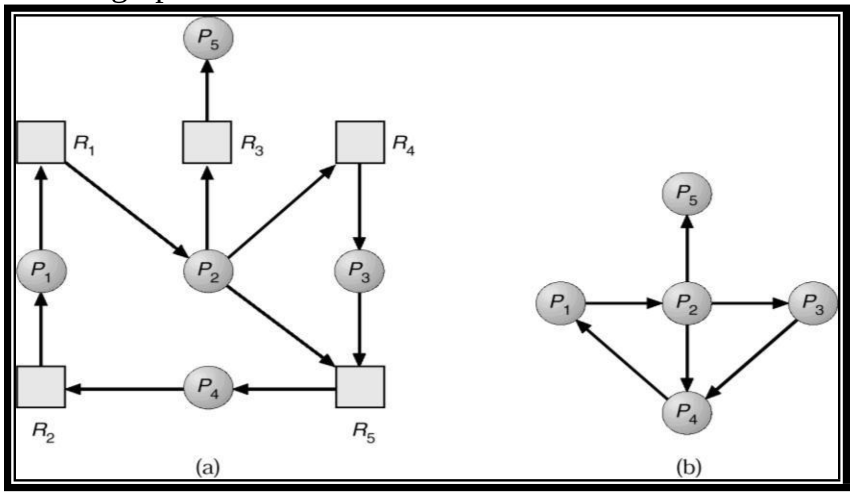
If a system does not employ either a deadlock-prevention or a deadlock avoidance algorithm, then a deadlock situation may occur. In this environment, the system must provide:

- An algorithm that examines the state of the system to determine whether a deadlock has occurred
- An algorithm to recover from the deadlock

Single Instance of Each Resource Type

- If all resources have only a single instance, then we can define a deadlock detection algorithm that uses a variant of the resource-allocation graph, called a *wait-for* graph.
- We obtain this graph from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges.
- More precisely, an edge from Pi to Pj in a wait-for graph implies that process Pi is waiting for process Pj to release a resource that Pi needs.

- An edge *Pi*→ Pj exists in a wait-for graph if and only if the corresponding resource allocation graph contains two edges Pi —> *Rq* and Rq → Pj for some resource Rq.
- A deadlock exists in the system if and only if the wait-for graph contains a cycle. To detect deadlocks, the system needs to *maintain* the wait-for graph and periodically *invoke an algorithm* that searches for a cycle in the graph.
- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where *n* is the number of vertices in the graph.



(a) Resource-allocation graph, (b) Corresponding wait-for graph.
(b)

Several Instances of a Resource Type

- The wait-for graph scheme is not applicable to a resource-allocation system with multiple instances of each resource type.
- We turn now to a deadlock detection algorithm that is applicable to such a system.
- The algorithm employs several time-varying data structures that are similar to those used in the
- banker's algorithm :
  - **Available.** A vector of length *m* indicates the number of available resources of each type.
  - **Allocation.** An *n* x *m* matrix defines the number of resources of each type currently allocated to each process.
  - **Request.** An *n* x *in* matrix indicates the current request of each process. If *Request[i][j]* equals *k*, then process Pi is requesting *k* more instances of resource type *Rj.*
- The detection algorithm described here simply investigates every possible allocation sequence for the processes that remain to be completed.

1. Let *Work* and *Finish* be vectors of length *in* and *n,* respectively. Initialize *Work = Available.* For *i* = 0 , 1 , . . . , *n-1,* if *Allocation*$_i$!=0, then *Finish[i]= false;* otherwise, *FinisH[i] = true.*
2. Find an index *i* such that both
   a. *Finish[i] =false*
   b. *Request*$_i$ < *Work*

If no such / exists, go to step 4.

3. *Work - Work + Allocation$_1$*
   Finish[i] = true
   Go to step 2.
4. If *Finish[i] == false,* for some i, 0 < =i < *n,* then the system is in a deadlocked state. Moreover, if *Finish[i] == false,* then process *Pi* is deadlocked.

- This algorithm requires an order of m x n^2 operations to detect whether the system is in a deadlocked state.
- To illustrate this algorithm, we consider a system with five processes *P0* through P4 and three resource types *A, B,* and C.
- Resource type *A* has seven instances, resource type B has two instances, and resource type C has six instances. Suppose that, at time To, we have the following resource-allocation state:

	Allocation	*Request*	*Available*
	A B C	*A B C*	*A B C*
P_0	*0 1 0*	*0 0 0*	*0 0 0*
P_1	*2 0 0*	*2 0 2*	
P_2	*3 0 3*	*0 0 0*	
P_3	*2 1 1*	*1 0 0*	
P_4	*0 0 2*	*0 0 2*	

- Sequence <P_0, P_2, P_3, P_1, P_4> will result in Finish[i] = true for all i.
- Suppose P_2 requests an additional instance of type C.

	Request
	A B C
P_0	*0 0 0*
P_1	*2 0 1*
P_2	*0 0 1*
P_3	*1 0 0*
P_4	*0 0 2*

- State of system?
  - Can reclaim resources held by process P_0, but insufficient resources to fulfill other processes; requests.
  - Deadlock exists, consisting of processes P_1, P_2, P_3, and P_4.

Detection-Algorithm Usage

- When should we invoke the detection algorithm? The answer depends on two factors:
  - How *often* is a deadlock likely to occur?
  - How *many* processes will be affected by deadlock when it happens?
- If deadlocks occur frequently, then the detection algorithm should be invoked frequently.
- Resources allocated to deadlocked processes will be idle until the deadlock can be broken.
- In addition, the number of processes involved in the deadlock cycle may grow.

- Deadlocks occur only when some process makes a request that cannot be granted immediately.
- This request may be the final request that completes a chain of waiting processes. In the extreme, we can invoke the deadlock detection algorithm every time a request for allocation cannot be granted immediately.
- In this case, we can identify not only the deadlocked set of processes but also the specific process that "caused" the deadlock.
- If there are many different resource types, one request may create many cycles in the resource graph, each cycle completed by the most recent request and "caused" by the one identifiable process.
- Of course, if the deadlock-detection algorithm is invoked for every resource request, this will incur a considerable overhead in computation time.
- A less expensive alternative is simply to invoke the algorithm at less frequent intervals for example, once per hour or whenever CPU utilization drops below 40 percent.
- If the detection algorithm is invoked at arbitrary points in time, there may be many cycles in the resource graph. In this case, we would generally not be able to tell which of the many deadlocked processes "caused" the deadlock.

Recovery From Deadlock

- When a detection algorithm determines that a deadlock exists, several alternatives are available.
- One possibility is to inform the operator that a deadlock has occurred and to let the operator deal with the deadlock manually.
- Another possibility is to let the system *recover* from the deadlock automatically.
- There are two options for breaking a deadlock.
- One is simply to abort one or more processes to break the circular wait.
- The other is to preempt some resources from one or more of the deadlocked processes.

Process Termination

- To eliminate deadlocks by aborting a process, we use one of two methods.
- In both methods, the system reclaims all resources allocated to the terminated processes.
- **Abort all deadlocked processes.** This method clearly will break the deadlock cycle, but at great expense; the deadlocked processes may have computed for a long time, and the results of these partial computations must be discarded and probably will have to be recomputed later.
- **Abort one process at a time until the deadlock cycle is eliminated.** This method incurs considerable overhead, since, after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked.
- Aborting a process may not be easy. If the process was in the midst of updating a file, terminating it will leave that file in an incorrect state.
- Similarly, if the process was in the midst of printing data on a printer, the system must reset the printer to a correct state before printing the next job.

- If the partial termination method is used, then we must determine which deadlocked process (or processes) should be terminated. This determination is a policy decision, similar to CPU-scheduling decisions.
- The question is basically an economic one; we should abort those processes whose termination will incur the minimum cost.
- Unfortunately, the term *minimum cost* is not a precise one.
- Many factors may affect which process is chosen, including:
  - What the priority of the process is.
  - How long the process has computed and how much longer the process will compute before completing its designated task.
  - How many and what type of resources the process has used (for example, whether the resources are simple to preempt)
  - How many more resources the process needs in order to complete.
  - How many processes will need to be terminated
  - Whether the process is interactive or batch

Resource Preemption

- To eliminate deadlocks using resource preemption, we successively preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken.
- If preemption is required to deal with deadlocks, then three issues need to be addressed:
  1. **Selecting a victim**. Which resources and which processes are to be preempted? As in process termination, we must determine the order of preemption to minimize cost. Cost factors may include such parameters as the number of resources a deadlocked process is holding and the amount of time the process has thus far consumed during its execution.
  2. **Rollback.** If we preempt a resource from a process, what should be done with that process? Clearly, it cannot continue with its normal execution; it is missing some needed resource. We must roll back the process to some safe state and restart it from that state.

     Since, in general, it is difficult to determine what a safe state is, the simplest solution is a total rollback: Abort the process and then restart it. Although it is more effective to roll back the process only as far as necessary to break the deadlock, this method requires the system to keep more information about the state of all running processes.
  3. **Starvation**. How do we ensure that starvation will not occur? That is, how can we guarantee that resources will not always be preempted from the same process?

Revised questions

MAY-2011

6. (a) Give Banker's algorithm and explain with suitable example. 8

6. (a) Discuss about the deadlock prevention approach 8

6. (a) What is safe state and unsafe state? Discuss about the deadlock avoidance. 8

6. (a) What is a deadlock? List the necessary conditions and Explain about resource allocation graph. 8

MAY-2010

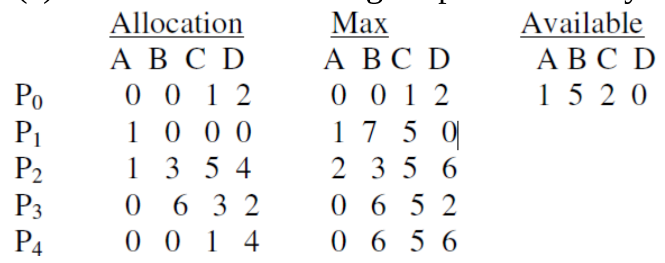
6. What is Safe state? Write the Bankers algorithm and explain it with the help of an example.[16]

6. (a) What is Resource allocation graph? How Resource allocation graph can be used in the context of Deadlocks.
(b) How Deadlocks can be prevented considering the four necessary conditions. [6+10]

6. What is Deadlock? What are the necessary conditions for deadlock? What are the overheads associated with Deadlock prevention and Deadlock avoidance algorithms. [16]

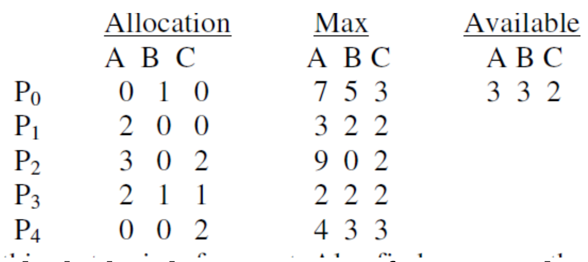
NOV-2012

6. (a) Consider the following snapshot of the system



	Allocation	Max	Available
	A B C D	A B C D	A B C D
P_0	0 0 1 2	0 0 1 2	1 5 2 0
P_1	1 0 0 0	1 7 5 0	
P_2	1 3 5 4	2 3 5 6	
P_3	0 6 3 2	0 6 5 2	
P_4	0 0 1 4	0 6 5 6	

Find whether this system is safe or not .Also find sequence that satisfies safety requirement.
(b) What are the four conditions that hold simultaneously in a system for the deadlock situation to arise?
6. (a) Consider a system with five processes P0 through P4 and three resource types A,B,C. Resource type A has 10 instances ,resource type B has 5 instances and resource type C has 7 instances . Suppose that, at time T0 the following snapshot of the system has been Taken



	Allocation	Max	Available
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Find whether this system is safe or not .Also find sequence that satisfies safety requirement.
(b) What is the optimistic assumption made in deadlock-detection algorithm? How could this assumption be violated?
6. (a) Explain in detail the banker's algorithm.
(b) Explain about deadlock characterization in detail.

6. (a) Explain the procedure for eliminating deadlocks using resource pre-emption.
(b) How can we prevent the occurrence of deadlocks? Discuss in brief.

FILE SYSTEM

Syllabus

File system Interface
The concept of a file
Access Methods
Directory structure
File system mounting
File sharing
Protection

File system Implementation
File system structure
File system Implementation
Directory Implementation
Allocation methods
Free space management

File system structure
File system Implementation
Directory Implementation
Allocation methods
Free space management

File system Interface

Introduction

- ***File system*** provides the mechanism for on-line storage of and access to both data and programs of the operating system and all the users of the computer system.
- The file system consists of two distinct parts: a collection *of files,* each storing related data, and a *directory structure,* which organizes and provides information about all the files in the system.

File Concept

- Computers can store information on various storage media, such as magnetic disks, magnetic tapes, and optical disks.
- The operating system abstracts from the physical properties of its storage devices to define a logical storage unit*, the file.*
- Files are mapped by the operating system onto physical devices.
- These storage devices are usually nonvolatile, so the contents are persistent through power failures and system reboots.
- A file is a named collection of related information that is recorded on secondary storage.

- Data cannot be written to secondary storage unless they are within a file.
- Data files may be numeric, alphabetic, alphanumeric, or binary.
- In general, a file is a sequence of bits, bytes, lines, or records, the meaning of which is defined by the file's creator and user.
- The information in a file is defined by its creator.
- Many different types of information may be stored in a file—source programs, object programs, executable programs, numeric data, text, payroll records, graphic images, sound recordings, and so on.
- A file has a certain defined structure, which depends on its type.
- A *text* file is a sequence of characters organized into lines (and possibly pages).
- A *source* file is a sequence of subroutines and functions, each of which is further organized as declarations followed by executable statements.
- An *object* file is a sequence of bytes organized into blocks understandable by the system's linker.
- An *executable* file is a series of code sections that the loader can bring into memory and execute.

File Attributes

- A file is named, for the convenience of its human users, and is referred to by its name.
- A name is usually a string of characters, such as example.c.
- Some systems differentiate between uppercase and lowercase characters in names, whereas other systems do not.
- When a file is named, it becomes independent of the process, the user, and even the system that created it.
- For instance, one user might create the file example.c, and another user might edit that file by specifying its name.
- The file's owner might write the file to a floppy disk, send it in an e-mail, or copy it across a network, and it could still be called example.c on the destination system.
- A file's attributes vary from one operating system to another but typically consist of these:
  Name. The symbolic file name is the only information kept in human readable form,.
  Identifier. This unique tag, usually a number, identifies the file within the file system; it is the non-human-readable name for the file.
  Type. This information is needed for systems that support different types of files.
  Location. This information is a pointer to a device and to the location of the file on that vice.
  Size. The current size of the file (in bytes, words, or blocks) and possibly the maximum allowed size are included in this attribute.
  Protection. Access-control information determines who can do reading, writing, executing, and so on.
  Time, date, and user identification. This information may be kept for creation, last modification, and last use. These data can be useful for protection, security, and usage monitoring.
- The information about all files is kept in the directory structure, which also resides on secondary storage.
- Typically, a directory entry consists of the file's name and its unique identifier.

- The identifier in turn locates the other file attributes. It may take more than a kilobyte to record this information for each. file.

File Operations

- A file is an **abstract data type.** To define a file properly, we need to consider the operations that can be performed on files.
- The operating system can provide system calls to create, write, read, reposition, delete, and truncate files.
- Operating system performs six basic file operations. It should then be easy to see how other, similar operations, such as renaming a file, can be implemented.

  1. **Creating a file.** Two steps are necessary to create a file. First, space in the file system must be found for the file. Entry for the new file must be made in the directory.
  2. **Writing a file.** To write a file, we make a system call specifying both the name of the file and the information to be written to the file. Given the name of the file, the system searches the directory to find the file's location. The system must keep a *write* pointer to the location in the file where the next write is to take place. The write pointer must be updated whenever a write occurs.
  3. **Reading a file.** To read from a file, we use a system call that specifies the name of the file and where the next block of the file should be put. Again, the directory is searched for the associated entry, and the system needs to keep a *read* pointer to the location in the file where the next read is to take place. Once the read has taken place, the read pointer is updated. Because a process is usually either reading from or writing to a file, the current operation location can be kept as a per-process **current file-position pointer.** Both the read and write operations use this same pointer, saving space and reducing system complexity.
  4. **Repositioning within** a **file.** The directory is searched for the appropriate entry, and the current-file-position pointer is repositioned to a given value. Repositioning within a file need not involve any actual I/O. This file operation is also known as a file *seek.*
  5. **Deleting a file.** To delete a file, we search the directory for the named file. Having found the associated directory entry, we release all file space, so that it can be reused by other files, and erase the directory entry.
  6. **Truncating a file.** The user may want to erase the contents of a file but keep its attributes. Rather than forcing the user to delete the file and then recreate it, this function allows all attributes to remain unchanged—except for file length—but lets the tile be reset to length zero and its file space released.

- Most of the file operations mentioned involve searching the directory for the entry associated with the named file. To avoid this constant searching, many systems require that an open() system call be made before a file is first used actively.
- The operating system keeps a small table, called the **open-file table,** containing information about all open files.
- When a file operation is requested, the file is specified via an index into this table, so no searching is required.

- When the file is no longer being actively used, it is *closed* by the process, and the operating system removes its entry from the open-file table, create and delete are system calls that work with closed rather than open files.
- Some systems implicitly open a file when the first reference to it is made.
- The file is automatically closed when the job or program that opened the file terminates.
- Most systems, however, require that the programmer open a file explicitly with the open() system call before that file can be used.
- The open() operation takes a file name and searches the directory, copying the directory entry into the open-file table.
- The open() call can also accept access mode information—create, read-only, read—write, append-only, and so on.
- This mode is checked against the file's permissions. If the request mode is allowed, the file is opened for the process.
- The open() system call typically returns a pointer to the entry in the open-file table.
- This pointer, not the actual file name, is used in all I/O operations, avoiding any further searching and simplifying the system-call interface.
- The implementation of the open () and close() operations is more complicated in an environment where several processes may open the file at the same time.
- This may occur in a system where several different applications open the same file at the same time.
- Typically, the operating system uses two levels of internal tables: a per-process table and a system-wide table.
- The per process table tracks all files that a process has open. Stored in this table is information regarding the use of the file by the process.
- For instance, the current file pointer for each file is found here. Access rights to the file and accounting information can also be included.
- Each entry in the per-process table in turn points to a system-wide open-file table.
- The system-wide table contains process-independent information, such as the location of the file on disk, access dates, and file size.
- Once a file has been opened by one process, the system-wide table includes an entry for the file.
- When another process executes an open() call, a new entry is simply added to the process's open-file table pointing to the appropriate entry in the system wide table.
- Typically., the open-file table also has an *open count* associated with each file to indicate how many processes have the file open.
- Each close() decreases this *open count,* and when the *open count* reaches zero, the file is no longer in use, and the file's entry is removed from the open-file table.
- In summary, several pieces of information are associated with an open file.
  - **File pointer**. On systems that do not include a file offset as part of the read() and write () system calls, the system must track the last read write location as a current-file-position pointer. This pointer is unique to each process operating on the file and therefore must be kept separate from the on-disk file attributes.
  - **File-open count.** As files are closed, the operating system must reuse its open-file table entries, or it could run out of space in the table. Because multiple processes may have opened a file, the system must wait for the last file to close before removing the open-file table entry.

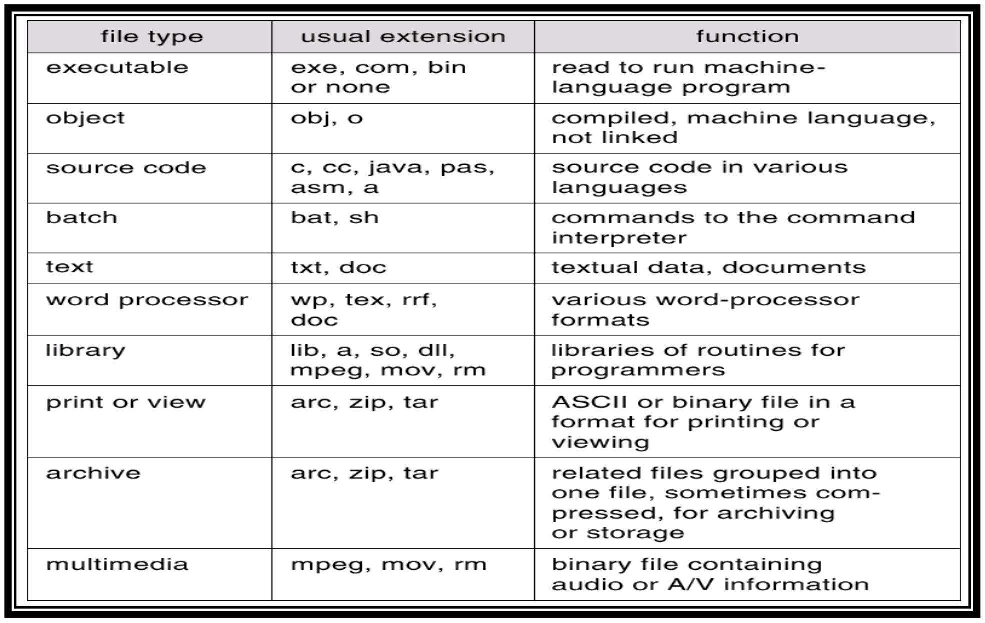
The file-open counter tracks the number of opens and closes and reaches zero on the last close. The system can then remove the entry.

- **Disk location of the file.** Most file operations require the system to modify data within the file. The information needed to locate the file on disk is kept in memory so that the system does not have to read it from disk for each operation.
- **Access rights.** Each process opens a file in an access mode. This information is stored on the per-process table so the operating system can allow or deny subsequent I/O requests.

File Types

- When we design a file system—, an entire operating system—we always consider whether the operating system should recognize and support file types.
- If an operating system recognizes the type of a file, it can then operate on the file in reasonable ways.
- For example, a common mistake occurs when a user tries to print the binary-object form of a program. This attempt normally produces garbage.
- Common technique for implementing file types is to include the type as part of the file name.
- The name is split into two parts—a name and an *extension,* usually separated by a period character.
- In this way, the user and the operating system can tell from the name alone what the type of a file is.
- For example, most operating systems allow users to specify file names as a sequence of characters followed by a period and terminated by an extension of additional characters.
- File name examples include *resume.doc, Scrver.java,* and *ReaderThread.c.*
- The system uses the extension to indicate the type of the file and the type of operations that can be done on that file. Only a file with a *.com, .exe,* or *.bat* extension can be *executed,* for instance.
- The *.com* and *.exe* files are two forms of binary executable files, whereas a *.bat* file is a batch file containing, in ASCII format, commands to the operating system.
- MS-DOS recognizes only a few extensions, but application programs also use extensions to indicate file types in which they are interested.
- For example, assemblers expect source files to have an *.asm* extension, and the Microsoft Word processor expects its files to end with a *.doc* extension.
- These extensions are not required, so a user may specify a file without the extension and the application will look for a file with the given name and the extension it expects. Because these extensions are not supported by the operating system, they can be considered as "hints" to the applications that operate on them.

Common file types.



file type	usual extension	function
executable	exe, com, bin or none	read to run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rrf, doc	various word-processor formats
library	lib, a, so, dll, mpeg, mov, rm	libraries of routines for programmers
print or view	arc, zip, tar	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, rm	binary file containing audio or A/V information

<u>File Structure</u>

- File types also can be used to indicate the internal structure of the file.
- Os requires that an executable file have a specific structure so that it can determine where in memory to load the file and what the location of the first instruction is.
- If the operating system defines five different file structures, it needs to contain the code to support these file structures.
- In addition, every file may need to be definable as one of the file types supported by the OS.
- Some operating systems support minimal number of file structures.
- Ex: UNIX, MS-DOS, and others.
- UNIX considers each file to be a sequence of 8-bit bytes; no interpretation of these bits is made by the operating system.
- This scheme provides maximum flexibility but little support. Each application program must include its own code to interpret an input file as to the appropriate structure.
- However, all operating systems must support at least one structure—that of an executable file—so...that the system is able to load and run programs.
- The Macintosh operating system also supports a minimal number of file structures.
- It expects files to contain two parts: a resource fork and a data fork.
- The resource fork contains information of interest to the user.
- For instance, it holds the labels of any buttons displayed by the program. A foreign user may want to re-label these buttons in his own language, and the Macintosh operating system provides tools to allow modification of the data in the resource fork.
- The data fork contains program code or data—the traditional file contents. To accomplish the same task on a UNIX or MS-DOS system, the programmer would need to change and recompile the source code, unless she created her own user-changeable data file.

<u>Internal File Structure</u>

- Locating an offset within a file can be complicated for the operating system. Disk systems typically have a well-defined block size determined by the size of a sector.
- All disk I/O is performed in units of one block (physical record), and all blocks are the same size.
- It is unlikely that the physical record size will exactly match the length of the desired logical record.
- Logical records may even vary in length. Packing a number of logical records into physical blocks is a common solution to this problem.
- For example, the UNIX operating system defines all files to be simply streams of bytes.
- Each byte is individually addressable by its offset from the beginning (or end) of the file.
- In this case, the logical record size is 1 byte.
- The file system automatically packs and unpacks bytes into physical disk blocks—say, 512 bytes per block—as necessary.
- The logical record size, physical block size, and packing technique determine how many logical records are in each physical block.
- The packing can be done either by the user's application program or by the operating system.
- In either case, the file may be considered to be a sequence of blocks.
- All the basic I/O functions operate in terms of blocks.
- The conversion from logical records to physical blocks is a relatively simple software problem.
- Because disk space is always allocated in blocks, some portion of the last block of each file is generally wasted.
- If each block were 512 bytes, for example, then a file of 1,949 bytes would be allocated four blocks (2,048 bytes); the last 99 bytes would be wasted.
- The waste incurred to keep everything in units of blocks (instead of bytes) is internal fragmentation.
- All file systems suffer from internal fragmentation; the larger the block size, the greater the internal fragmentation.

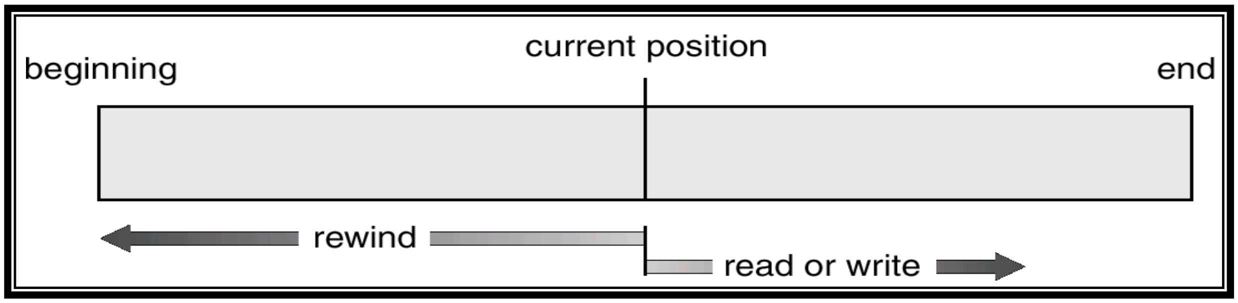
Access Methods

- Files store information.
- When it is used, this information must be accessed and read into computer memory.
- The information in the file can be accessed in several ways. Those are
    1. Sequential Access
    2. Direct Access
    3. Indirect Access

Sequential Access

- The simplest access method is sequential access.
- Information in the file is processed in order, one record after the other.
- This mode of access is by far the most common; for example, editors and compilers usually access files in this fashion.
- Reads and writes make up the bulk of the operations on a file.

- A read operation—*read next*—reads the next portion of the file and automatically advances a file pointer, which tracks the I/O location.
- Similarly, the write operation—*write next*—appends to the end of the file and advances to the
- end of the newly written material (the new end of file).
- Such a file can be reset to the beginning; and on some systems, a program .may be able to skip forward or backward *n* records for some integer *n*—perhaps only for *n* = 1.
- Sequential access, which is shown in below Figure is based on a tape model of a file and works as well on sequential-access devices as it does on random-access ones.

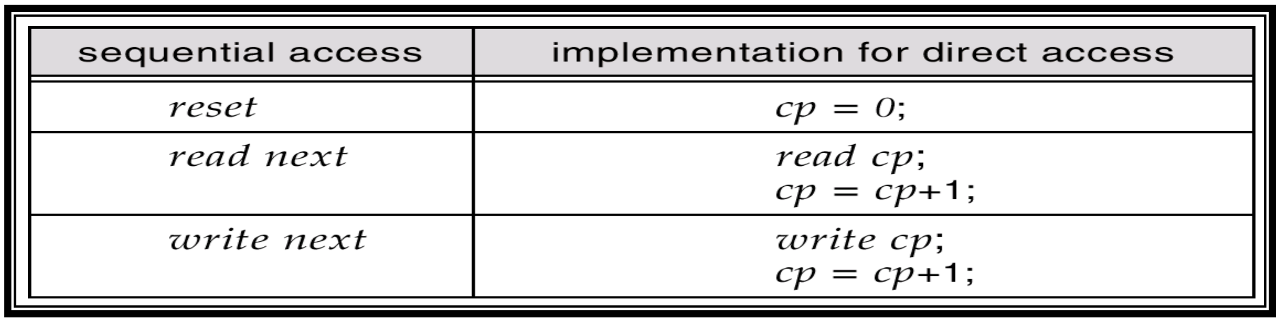


Sequential-access file.

Direct Access

- Another method is **direct access** (or **relative** access). A file is made up of fixed length **logical records** that allow programs to read and write records rapidly in no particular order. The direct-access method is based on a disk model of a file, since disks allow random access to any file block.
- For direct access, the file is viewed as a numbered sequence of blocks or records. Thus, we may read block 14, then read block 53, and then write block 7.
- There are no restrictions on the order of reading or writing for a direct-access file.
- Direct-access files are of great use for immediate access to large amounts of information.
- Databases are often of this type.
- Example: an airline-reservation system, we might store all the information about a particular flight (for example, flight 713) in the block identified by the flight number. Thus, the number of available seats for flight 713 is stored in block 713 of the reservation file.
- For the direct-access method, the file operations must be modified to include the block number as a parameter. Thus, we have *read n,* where *n* is the block number, rather than *read next,* and *write n* rather than *write next.*
- An alternative approach is to retain *read next* and *write next,* as with sequential access, and to add an operation *position file to n,* where *n* is the block number.
- Then, to effect a *read n,* we would, *position to n* and then *read next.*
- The block number provided by the user to the operating system is normally a **relative block number.**
- A relative block number is an index relative to the beginning of the file.
- Thus, the first relative block of the file is 0, the next is 1, and so on, even though the actual absolute disk address of the block may be 14703 for the first block and 3192 for the second.

- The use of relative block numbers allows the operating system to decide where the file should be place and helps to prevent the user from accessing portions of the file system that may not be part of her file.
- How then does the system satisfy a request for record *N* in a file? Assuming we have a logical record length *L*, the request for record N is turned into an I/O request for *L* bytes starting at location *L* * (N) within the file . Since logical records are of a fixed size, it is also easy to read, write, or delete a record.
    - We can easily simulate sequential access on a direct-access file by simply keeping a variable *cp* that defines our current position, as

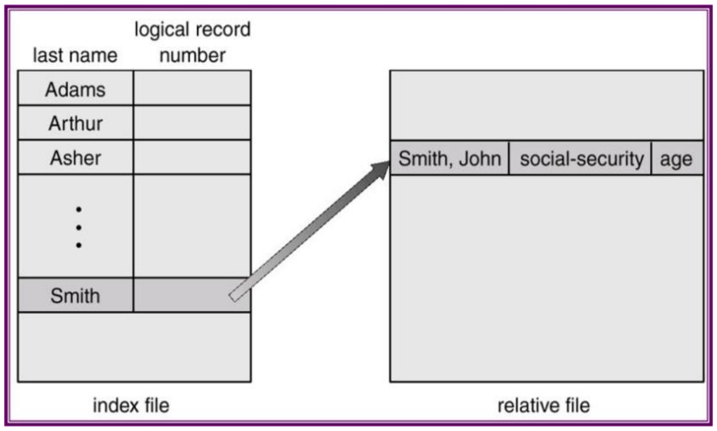


sequential access	implementation for direct access
reset	*cp = 0*;
read next	*read cp*; *cp = cp*+1;
write next	*write cp*; *cp = cp*+1;

Simulation of sequential access on a direct-access file.

Other Access Methods

- This method generally involves the construction of an index for the file.
- The index, like an index in the back of a book, contains pointers to the various blocks.
- To find a record in the file, we first search the index and then use the pointer to access the file directly and to find the desired record.
- For example, a retail-price file might list the universal product codes (UPCs) for items, with the associated prices. Each record consists of a 10-digit UPC and a 6-digit price, for a 16-byte record, if our disk has 1,024 bytes per block, we can store 64 records per block.
- A file of 120,000 records would occupy about 2,000 blocks (2 million bytes). By keeping the file sorted by UPC, we can define an index consisting of the first UPC in each block.
- This index would have 2,000 entries of 10 digits each, or 20,000 bytes, and thus could be kept in memory.



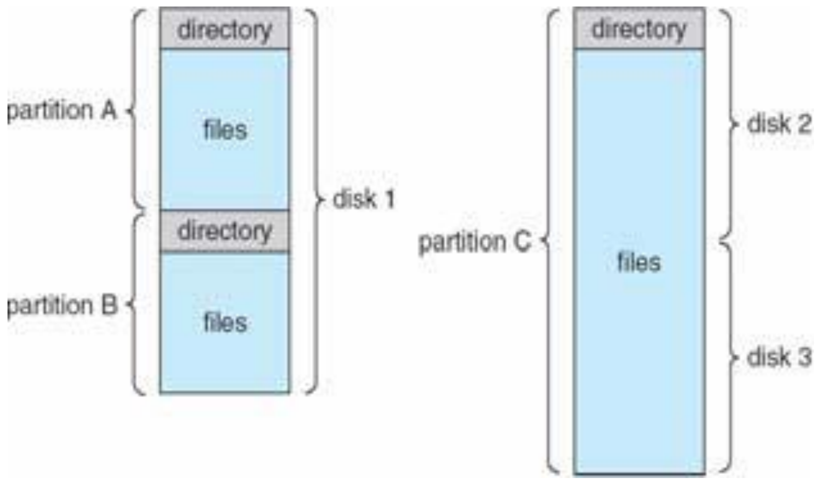
Example of index and relative files.

Directory Structure

- Systems may have zero or more file systems, and the file systems may be of varying types.
- For example, a typical Solaris system may have a few UFS file systems, a VFS file system, and some NFS file systems.
- The file systems of computers, then, can be extensive. Some systems store millions of files on terabytes of disk. To manage all these data, we need to organize them. This organization involves the use of directories.

Storage Structure

- A disk (or any storage device that is large enough) can be used in its entirety for a file system. Sometimes, though, it is desirable to place multiple file systems on a disk or to use parts of a disk for a file system and other parts for other things, such as swap space or unformatted (raw) disk space.
- These parts are known variously as **partitions, slices,** or (in the IBM world) **minidisks.**
- A file system can be created on each of these parts of the disk.
- The parts can also be combined to form larger structures known as **volumes,** and file systems can be created on these as well.
- we simply refer to a chunk of storage that holds a file system as a volume.
- Each volume can be thought of as a virtual disk. Volumes can also store multiple operating systems, allowing a system to boot and run more than one.
- Each volume that contains a file system must also contain information about the files in the system. This information is kept in entries in a **device directory** or **volume table of contents.**
- The device directory (more commonly known simply as a **directory)** records information—such as name, location, size, and type—for all files on that volume.



A typical file-system organization

Directory Overview

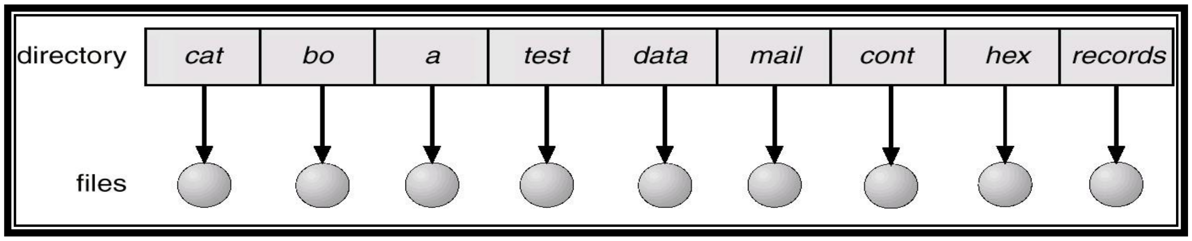
- The directory can be viewed as a symbol table that translates file names into their directory entries.
- The directory itself can be organized in many ways.
- We want to be able to insert entries, to delete entries, to search for a named entry, and to list all the entries in the directory.
- When considering a particular directory structure, we need to keep in mind the operations that are to be performed on a directory:
  1. **Search for a file**. We need to be able to search a directory structure to find the entry for a particular file. Since files have symbolic names and similar names may indicate a relationship between files, we may want to be able to find all files whose names match a particular pattern.
  2. **Create a file.** New files need to be created and added to the directory.
  3. **Delete a file.** When a file is no longer needed, we want to be able to remove it from the directory.
  4. **List a directory.** We need to be able to list the files in a directory and the contents of the directory entry for each file in the list.
  5. **Rename a file**. Because the name of a file represents its contents to its users, we must be able to change the name when the contents or use of the file changes. Renaming a file may also allow its position within the directory structure to be changed.
  6. **Traverse the file system.** We may wish to access every directory and every file within a directory structure. For reliability, it is a good idea to save the contents and structure of the entire file system at regular intervals.

- Common schemes for defining the logical structure of a directory.
  1. Single Level Directory
  2. Two Level Directory
  3. Tree structured Directory
  4. Acyclic graph Directory

5. General graph Directory

Single-Level Directory

- The simplest directory structure is the single-level directory.
- All files are contained in the same directory, which is easy to support and understand.



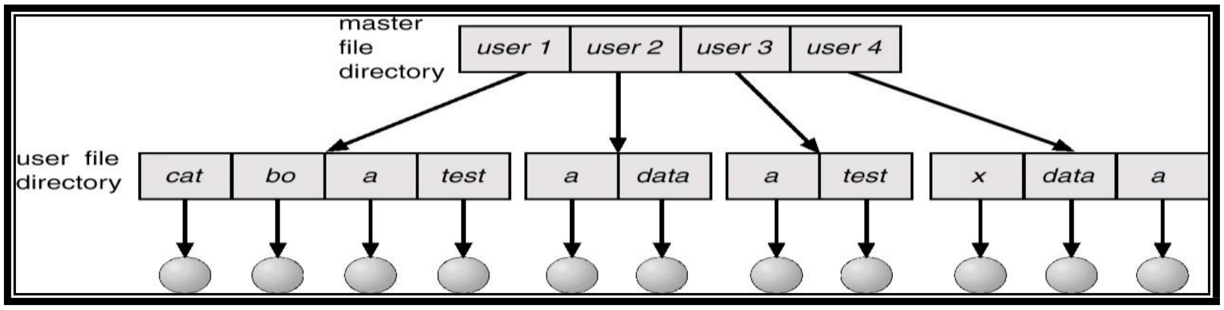
Single-level directory.

- A single-level directory has significant limitations, however, when the number of files increases or when the system has more than one user. Since all files are in the same directory, they must have unique names.
- If two users call their data file *test,* then the unique-name rule is violated.
- Even a single user on a single-level directory may find it difficult to remember the names of all the files as the number of files increases.

Two-Level Directory

- As we have seen, a single-level directory often leads to confusion of file names among different users.
- The standard solution is to create a *separate* directory for each user.
- In the two-level directory structure, each user has his own user file directory (UFD). The UFDs have similar structures, but each lists only the files of a single user.

- When a user job starts or a user logs in, the system's master file directory (MFD) is searched.
- The MFD is indexed by user name or account number, and each entry points to the UFD for that user .
- When a user refers to a particular file, only his own UFD is searched.
- Thus, different users may have files with the same name, as long as all the file names within each UFD are unique.
- To create a file for a user, the operating system searches only that user's UFD to ascertain whether another file of that name exists.
- To delete a file, the operating system confines its search to the local UFD; thus, it cannot accidentally delete another user's file that has the same name.



Two-level directory structure.

- The user directories themselves must be created and deleted as necessary.
- A special system program is run with the appropriate user name and account information.
- The program creates a new UFD and adds an entry for it to the UFD.
- The execution of this program might be restricted to system administrators.
- Although the two-level directory structure solves the name-collision problem, it still has disadvantages.
- This structure effectively isolates one user from another.
- Isolation is an advantage when the users are completely independent but is a disadvantage when the users *want* to cooperate on some task and to access one another's files.
- A two-level directory can be thought of as a tree, or an inverted tree, of height 2. The root of the tree is the MFD.
- Its direct descendants are the UFDs. The descendants of the UFDs are the files themselves. The files are the leaves of the tree.
- Specifying a user name and a file name defines a path in the tree from the root (the MFD) to a leaf (the specified file). Thus, a user name and a file name define a *path name.*
- Every file in the system has a path name.
- To name a file uniquely, a user must know the path name of the file desired. For example, if user A wishes to access her own test file named *test,* she can simply refer to *test.*
- To access the file named *test* of user B (with directory-entry name *userb),* however, she might have to refer to */userb/test.*

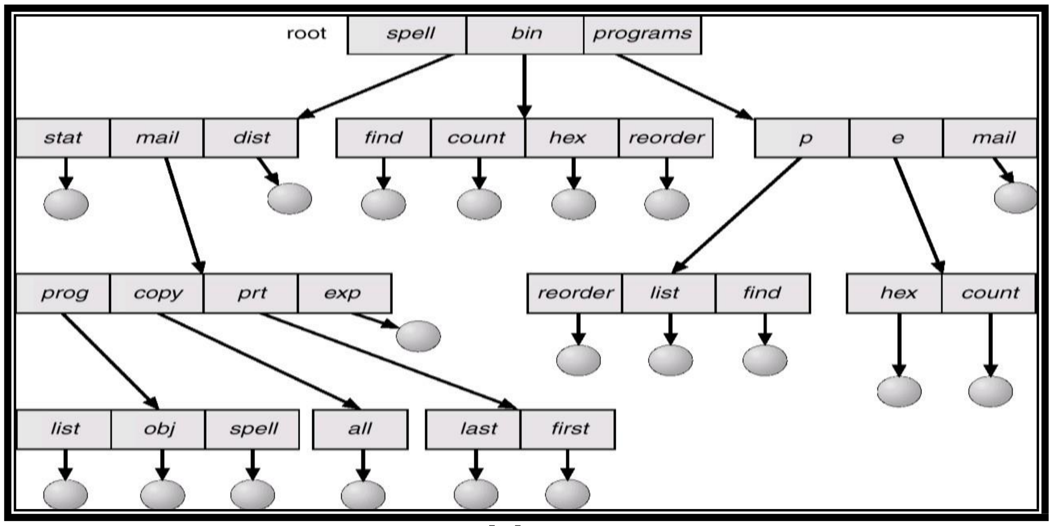
- Additional syntax is needed to specify the volume of a file. For instance, in MS-DOS a volume is specified by a letter followed by a colon. Thus, a file specification might be *C:\ userb\test.*
- *A* special case of this situation occurs with the system files. Programs provided as part of the system—loaders, assemblers, compilers, utility routines, libraries, and so on—are generally defined as files.
- When the appropriate commands are given to the operating system, these files are read by the loader and executed.
- Many command interpreters simply treat such a command as the name of a file to load and execute.
- As the directory system is defined presently, this file name would be searched for in the current UFD.

- One solution would be to copy the system files into each UFD. However, copying all the system files would waste an enormous amount of space. (If the system files require 5 MB, then supporting 12 users would require 5 x 12 = 60 MB just for copies of the System files.)
- Another solution - A special user directory is defined to contain the system files (for example, user 0). Whenever a file name is given to be loaded, the operating system first searches the local UFD. If the file is found, it is used.
- If it is not found, the system automatically searches the special user directory that contains the system files.
- The sequence of directories searched when a file is named is called the **search path.**

Tree-Structured Directories

- A Two level directory is also a tree with level two.
- This generalization allows users to create their own subdirectories and to organize their files accordingly.
- A tree is the most common directory structure.
- The tree has a root directory, and every file in the system has a unique path name.
- A directory (or subdirectory) contains a set of files or subdirectories.
- A directory is simply another file, but it is treated in a special way.
- All directories have the same internal format. One bit in each directory entry defines the entry as a file (0) or as a subdirectory (1).
- Special system calls are used to create and delete directories.
- Each process has a current directory. The **current directory** should contain most of the files that are of current interest to the process.
- When reference is made to a file, the current directory is searched.
- If a file is needed that is not in the current directory, then the user usually must either specify a path name or change the current directory to be the directory holding that file.
- To change directories, a system call is provided that takes a directory name as a parameter and uses it to redefine the current directory.
- The initial current directory of the login shell of a user is designated when the user job starts or the user logs in. The operating system searches the accounting file (or some other predefined location) to find an entry for this user (for accounting purposes).

- In the accounting file is a pointer to (or the name of) the user's initial directory.
- This pointer is copied to a local variable for this user that specifies the user's initial current directory.
- Path names can be of two types: *absolute* and *relative.*
- An **absolute path name** begins at the root and follows a path down to the specified file, giving the directory names on the path.
- A **relative path name** defines a path from the current directory. For example, in the tree-structured file system.

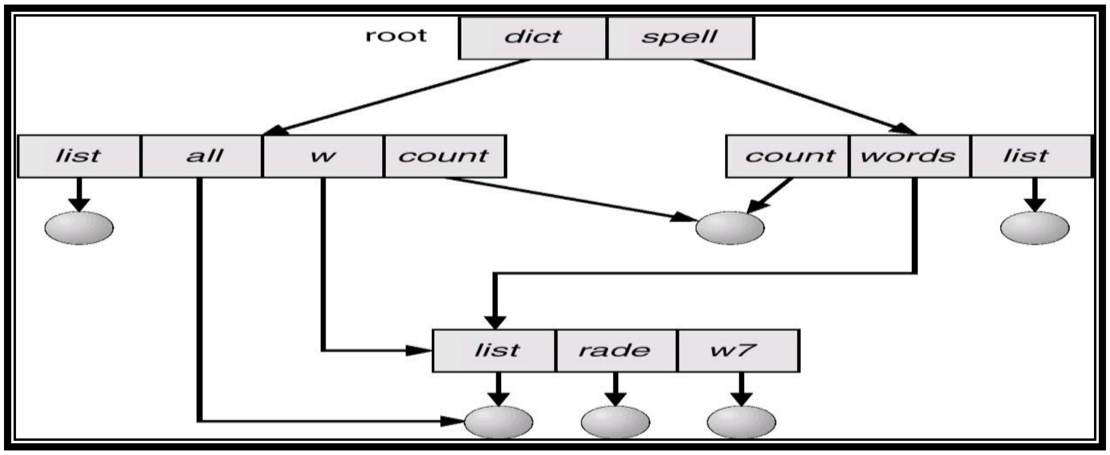


Tree-structured directory structure.

- If the current directory is *root/spell'/mail,* then the relative path name *prt/first* refers to the same file as does the absolute path name *root/spell/mail/prt/first.*
- *Handling deletion of a Directory:*
  - If a directory is empty, its entry in the directory is simply be deleted.
  - Suppose the directory to be deleted is not empty but contains several files or subdirectories.
  - One of two approaches can be taken. Some systems, such as MS-DOS, will not delete a directory unless it is empty. Thus, to delete a directory, the user must first delete all the files in that directory. If any subdirectories exist, this procedure must be applied recursively to them, so that they can be deleted also.
  - An alternative approach, such as that taken by the UNIX rm command, is to provide an option: When a request is made to delete a directory, all that directory's files and subdirectories are also to be deleted.

Acyclic-Graph Directories

- Consider two programmers who are working on a joint project. The files associated with that project can be stored in a subdirectory, separating them from other projects and files of the two programmers.
- But since both programmers are equally responsible for the project, both want the subdirectory to be in their own directories. The common subdirectory should be *shared.*
- *A* shared directory *or* file will exist in the file system in two (or more) places at once.
- A tree structure prohibits the sharing of files or directories.
- An **acyclic graph** —that is, a graph with no cycles—allows directories to share subdirectories and files.
- The *same* file or subdirectory may be in two different directories.
- The acyclic graph is a natural generalization of the tree-structured directory scheme.
- It is important to note that a shared file (or directory) is not the same as two copies of the file.

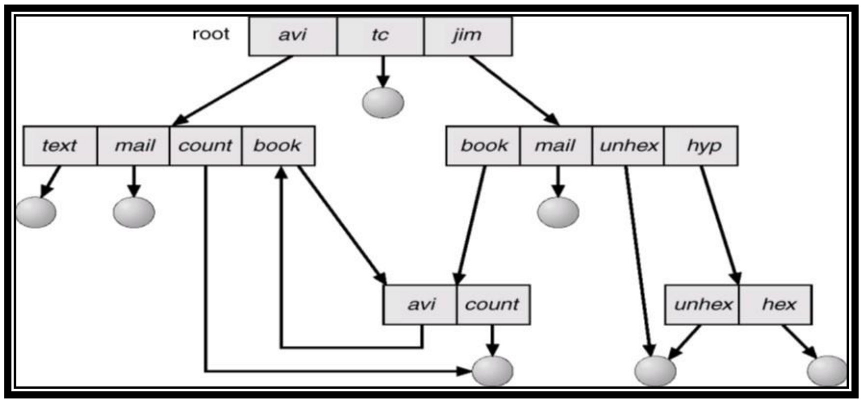


Acyclic-graph directory structure.

- When people are working as a team, all the files they want to share can be put into one directory.
- The UFD of each team member will contain this directory of shared files as a subdirectory.
- Shared files and subdirectories can be implemented in several ways.
  - A common way, In UNIX systems, creates a new directory entry called a link. A link is effectively a pointer to another file or subdirectory. For example, a link may be implemented as an absolute or a relative path name. When a reference to a file is made, we search the directory. If the directory entry is marked as a link, then the name of the real file is included in the link information.
  - Another common approach to implementing shared files is simply to duplicate all information about them in both sharing directories. Thus, both entries are identical and equal.
- *Problems:*
  - *Traversal* –T o find out a file, we may traverse shared structures more than once.
  - *Deletion:* One possibility is to remove the file whenever anyone deletes it, but this action may leave dangling pointers to the now-nonexistent file.
    - In a system where sharing is implemented the deletion of a link need not affect the original file; only the link is removed.
    - If the file entry itself is deleted, the space for the file is deallocated, leaving the links dangling. We can search for these links and remove them as well, but unless a list of the associated links is kept with each file, this search can be expensive.
    - Alternatively, we can leave the links until an attempt is made to use them.
    - Another approach to deletion is to preserve the file until all references to it are deleted. The file is deleted when its file-reference list is empty.

General Graph Directory

- A serious problem with using an acyclic-graph structure is ensuring that there are no cycles.



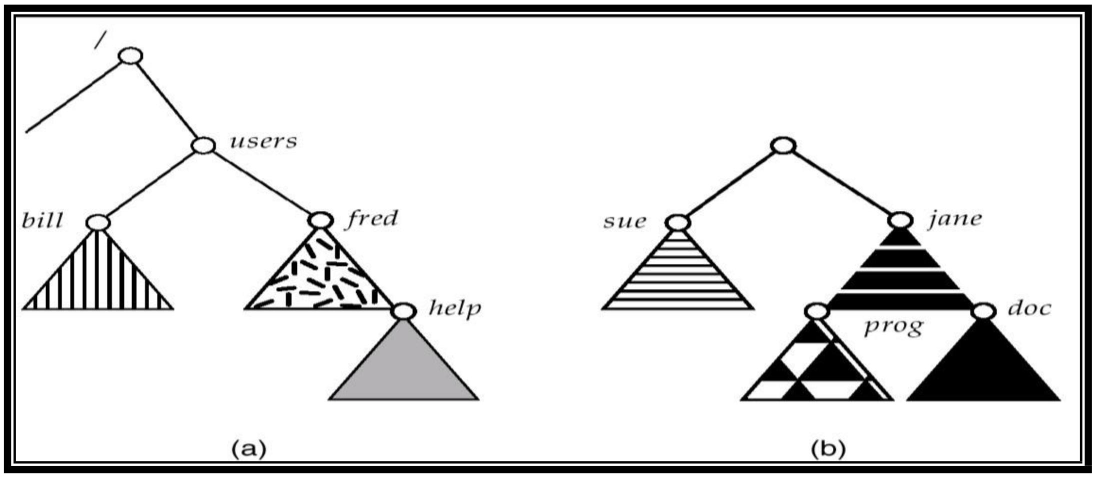
General graph directory.

- *Searching:*
- *In an acyclic graph*
  - We want to avoid traversing shared sections of an acyclic
  - graph twice, we have just searched a major
  - shared subdirectory for a particular file without finding it, we want to avoid
  - searching that subdirectory again;
- *If cycles are present*
  - A poorly designed algorithm might result in an infinite loop continually searching through the cycle and never terminating.
  - One solution is to limit arbitrarily the number of directories that will be accessed during a search.
- *Deletion*
- *In an acyclic graph*
  - A value of 0 in the reference count means that there are no more references to the file or directory, and the file can be deleted.
- *If cycles are present*
  - We use a garbage-collection scheme to determine when the last reference has been deleted and the disk space can be reallocated.
  - Garbage collection involves traversing the entire file system, marking everything that can be accessed.
  - Then, a second pass collects everything that is not marked onto a list of free space.
  - Same will be used for searching also.

File-System Mounting

- A file must be *opened* before it is used, a file system must be *mounted* before it can be available to processes on the system.
- The directory structure can be built out of multiple volumes, which must be mounted to make them available within the file-system name space.
- Mount procedure: The operating system is given the name of the device and the **mount point**—the location within the file structure where the file system is to be attached. Typically, a mount point is an empty directory.
- The operating system verifies that the device contains a valid file system. It does so by asking the device driver to read the device directory and verifying that the directory has the expected format.

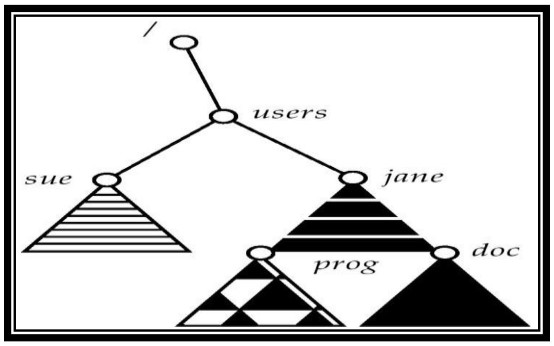
- Finally, the operating system notes in its directory structure that a file system is mounted at the specified mount point.



- *File system, (a) Existing system, (b) Unmounted volume.*

- To illustrate file mounting, consider the file system depicted in above Figure, where the triangles represent sub trees of directories that are of interest.
- An unmounted volume residing on */device'/disk.* At this point, only the files on the existing file system can be accessed.
- The below Figure shows the effects of mounting the volume residing on */device/disk* over */users.*

Mount point



Macintosh operating system.:

- Whenever the system encounters a disk for the first time (hard disks are found at boot time, and floppy disks are seen when they are inserted into the drive), the Macintosh operating system searches for a file system on the device.
- If it finds one, it automatically mounts the file system at the root level, adding a folder icon on the screen labeled with the name of the file system (as stored in the device directory).
- The user is then able to click on the icon and thus display the newly mounted file system.

File Sharing

Multiple Users

- When an operating system accommodates multiple users, the issues are file sharing, file naming, and file protection.

- The system can either allow a user to access the files of other users by default or require that a user specifically grant access to the files.
- To implement sharing and protection, the system must maintain more file and directory attributes than are needed on a single-user system.
- Most systems have evolved to use the concepts of file (or directory) *owner* (or *user)* and *group.*
- The owner is the user who can change attributes and grant access and who has the most control over the file.
- The group attribute defines a subset of users who can share access to the file.
- The owner and group IDs of a given file (or directory) are stored with the other file attributes.
- When a user requests an operation on a file, the user ID can be compared with the owner attribute to determine if the requesting user is the owner of the file.
- Likewise, the group IDs can be compared. The result indicates which permissions are applicable. The system then applies those permissions to the requested operation and allows or denies it.

Remote Fi!e Systems

- With the advent of networks, communication among remote computers became possible. Networking allows the sharing of resources spread across a campus or even around the world.
- One obvious resource to share is data in the form of files.
- The first implemented method involves manually transferring files between machines via programs like ftp.
- The second major method uses a **distributed file system (DFS)** in which remote directories is visible from a local machine.
- In some ways, the third method, the **World Wide Web,** is a reversion to the first. A browser is needed to gain access to the remote files, and separate operations (essentially a wrapper for ftp) are used to transfer files.
- Ftp is used for both anonymous and authenticated access. **Anonymous access** allows a user to transfer files without having an account on the remote system.
- The World Wide Web uses anonymous file exchange almost exclusively.
- DFS involves a much tighter integration between the machine that is accessing the remote files and the machine providing the files.

The Client- Server Model

- Remote file systems allow a computer to mount one or more file systems from one or more remote machines.
- In this case, the machine containing the files is the *server,* and the machine seeking access to the files is the *client.*
- The client-server relationship is common with networked machines. Generally, the server declares that a resource is available to clients and specifies exactly which resource (in this case, which files) and exactly which clients.
- A server can serve multiple clients, and a client can use multiple servers, depending on the implementation details of a given client-server facility.
- The server usually specifies the available files on a volume or directory level.

- Client identification is more difficult. A client can be specified by a network name or other identifier, such as an *IP address*, but these can be spoofed, or imitated.
- In the case of UNIX and its network file system (NFS), authentication takes place via the client networking information, by default. In this scheme, the user's IDs on the client and server must match.
- If they do not, the server will be unable to determine access rights to files.
- Access is thus granted or denied based on incorrect authentication information.
- Note that the NFS protocols allow many-to-many relationships. That is, many servers can provide files to many clients. In fact, a given machine can be both a server to other NFS clients and a client of other NFS servers.
- Once the remote file system is mounted, file operation requests are sent on behalf of the user across the network to the server via the DFS protocol.
- Typically, a file-open request is sent along with the ID of the requesting user.
- The server then applies the standard access checks to determine if the user has credentials to access the file in the mode requested.
- The request is either allowed or denied. If it is allowed, a file handle is returned to the client application, and the application then can perform read, write, and other operations on the file. The client closes the file when access is completed.

Distributed Information Systems

- To make client-server systems easier to manage, **distributed information** systems, also known as **distributed naming services,** provide unified access to the information needed for remote computing.
- The **domain name system (DNS)** provides host-name-to-network-address translations for the entire Internet (including the World Wide Web).
- Before DNIS became widespread, files containing the same information were sent via e-mail or ftp between all networked hosts.
- Other distributed information systems provide *user name/password/user ID/group ID* space for a distributed facility.
- Sun Microsystems introduced *yellow pages* (since renamed **network** information service, or NIS), and most of the industry adopted its use. It centralizes storage of user names, host names, printer information, and the like.
- Unfortunately, it uses unsecure authentication methods, including sending user passwords unencrypted (in *clear text)* and identifying hosts by IF address.
- In the case of Microsoft's **common** internet file system (CIFS), network information is used in conjunction with user authentication (user name and password) to create a network login that the server uses to decide whether to allow or deny access **to** a requested file system.
- The industry is moving toward use of the lightweight directory-access protocol (LDAP) as a secure distributed naming mechanism.

Consistency Semantics

- **Consistency semantics** represent an important criterion for evaluating any file system that supports file sharing.
- These semantics specify how multiple users of a system are to access a shared file simultaneously.
- In particular, they specify when modifications of data by one user will be observable by other users.
- These semantics are typically implemented as code with the file system.
- Consistency semantics are directly related to the process-synchronization algorithms.
- A series of file accesses (that is, reads and writes) attempted by a user to the same file is always enclosed between the open () and close () operations. The series of accesses between the open () and close () operations makes up a **file session.**
- Several prominent examples of consistency semantics.

*UNIX **Semantics***

- The UNIX file system uses the following consistency semantics:
  • Writes to an open file by a user are visible immediately to other users that have this file open.
  • One mode of sharing allows users to share the pointer of current location into the file. Thus, the advancing of the pointer by one user affects all sharing users. Here, a file has a single image that interleaves all accesses, regardless of their origin.
- In the UNIX semantics, a file is associated with a single physical image that is accessed as an exclusive resource.

Session Semantics

- The Andrew file system (AFS) uses the following consistency semantics:
  • Writes to an open file by a user are not visible immediately to other users that have the same file open.
  • Once a file is closed, the changes made to it are visible only in sessions starting later. Already open instances of the file do not reflect these changes.
- According to these semantics, a file may be associated temporarily with several (possibly different) images at the same time.

Immutable-Shared-Files Semantics

- A unique approach is that of **immutable shared files.** Once a file is declared as *shared* by its creator, it cannot be modified.
- An immutable file has two key properties: Its name may not be reused, and its contents may not be altered.
- Thus, the name of an immutable file signifies that the contents of the file are fixed.
- The implementation of these semantics in a distributed systemis simple, because the sharing is disciplined (read-only).

Protection

- When information is stored in a computer system, we want to keep it safe from physical damage *(reliability)* and improper access *(protection).*

- Reliability is generally provided by duplicate copies of files. Many computers have systems programs that automatically (or through computer-operator intervention) copy disk files to tape at regular intervals (once per day or week or month) to maintain a copy should a file system be accidentally destroyed.
- File systems can be damaged by hardware problems (such as errors in reading or writing), power surges or failures, head crashes, dirt, temperature extremes, and vandalism. Files may be deleted accidentally.

Types of Access

- The need to protect files is a direct result of the ability to access files. Systems that do not permit access to the files of other users do not need protection.
- Thus, we could provide complete protection by prohibiting access.
- Alternatively, we could provide free access with no protection. Both approaches are too extreme for general use. What is needed is **controlled access.**
- Protection mechanisms provide controlled access by limiting the types of file access that can be made.
- Access is permitted or denied depending on several factors, one of which is the type of access requested.
- Several different types of operations may be controlled:
  - Read. Read from the file.
  - Write. Write or rewrite the file.
  - Execute. Load the file into memory and execute it.
  - Append. Write new information at the end of the file.
  - Delete. Delete the file and tree its space for possible reuse.
  - List. List the name and attributes of the file.
- Other operations, such as renaming, copying, and editing the file, may also be controlled.

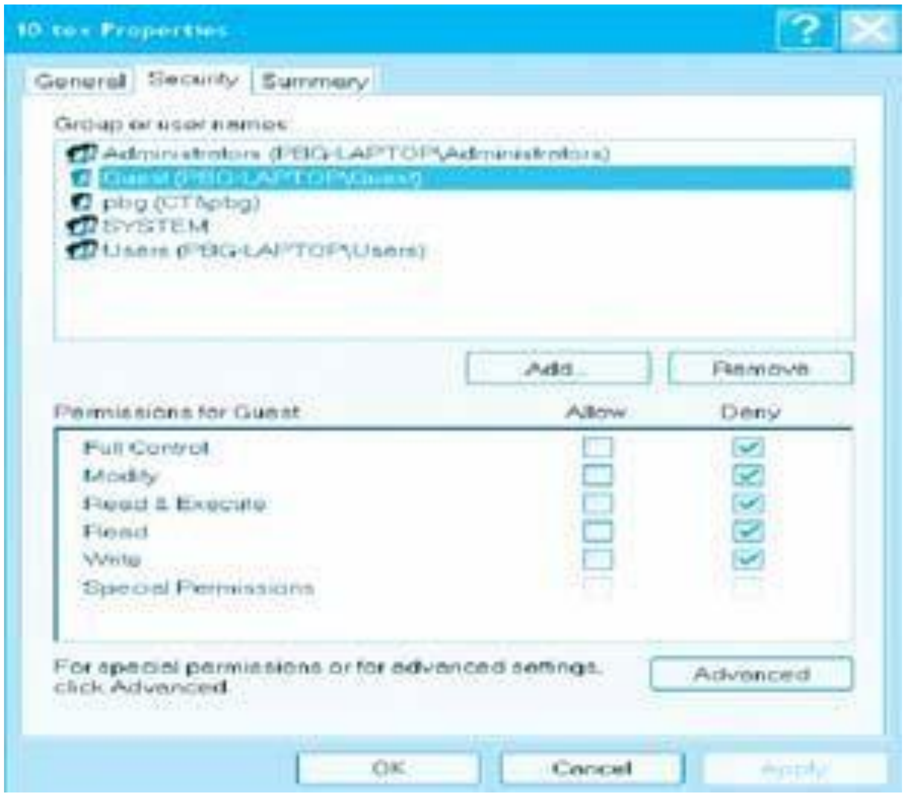
Access Control

- The most common approach to the protection problem is to make access dependent on the identity of the user.
- Different users may need different types of access to a. file or directory.
- The most general scheme to implement identity dependent access is to associate with each file and directory an access-control list (ACL) specifying user names and the types of access allowed for each user.
- When a user requests access to a particular file, the operating system checks the access list associated with that file.
- If that user is listed for the requested access, the access is allowed. Otherwise, a protection violation occurs, and the user job is denied access to the file.
- The main problem with access lists is their length. If we want to allow everyone to read a file, we must list all users with read access.
- This technique has two undesirable consequences:
- Constructing such a list may be **complicated**, if we do not know in advance the list of users in the system.

- The directory entry, previously of fixed size, now needs to be of variable size, resulting in more complicated space management.
- These problems can be resolved by use of a condensed version of the access list.
- To condense the length of the access-control list, many systems recognize three classifications of users in connection with each file:
  - • Owner. The user who created the file is the owner.
  - • Group. A set of users who are sharing the file and need similar access is a group, or work group.
  - • Universe. All other users in the system constitute the universe.
- The most common recent approach is to combine access-control lists with the more general (and easier to implement) owner, group, and universe access control scheme.
- For example, consider a person, Sara, who is writing a new book. She has hired three graduate students (Jim, Dawn, and Jill) to help with the project.
- The text of the book is kept in a file named *book.* The protection associated with this file is as follows:
  - Sara should be able to invoke all operations on the file.
  - Jim, Dawn, and Jill should be able only to read and write the file; they should not be allowed to delete the file.
  - All other users should be able to read, but not write, the file.
- To achieve such protection, we must create a new group—say, *text*— with members Jim, Dawn, and Jill. The name of the group, *text,* must then be associated with the file *book,* and the access rights must be set in accordance with the policy we have outlined.
- Only three fields are needed to define protection. Often, each field is a collection of bits, and each bit either allows or prevents the access associated with it.
- For example, the UNIX system defines three fields of 3 bits each—rwx, where r controls read access, w controls write access, and x controls execution.
- A separate field is kept for the file owner, for the file's group, and for all other users. In this scheme, nine bits per file are needed to record protection information.
- Thus, tor our example, the protection fields for the file *book* are as follows:
  - ❖ Mode of access: read, write, execute
- ❖ Three classes of users

			RWX
a) owner access	7	$\Rightarrow$	111
			RWX
b) group access	6	$\Rightarrow$	1 1 0
			RWX
c) public access	1	$\Rightarrow$	0 0 1

- ❖ Ask manager to create a group (unique name), say G, and add some users to the group.
- ❖ For a particular file (say game) or subdirectory, define an appropriate access.



10.tex Properties

General | Security | Summary

Group or user names:

- Administrators (PBG-LAPTOP\Administrators)
- Guest (PBG-LAPTOP\Guest)
- pbg (CTI\pbg)
- SYSTEM
- Users (PBG-LAPTOP\Users)

Permissions for Guest

	Allow	Deny
Full Control	☐	✓
Modify	☐	✓
Read & Execute	☐	✓
Read	☐	✓
Write	☐	✓
Special Permissions		

For special permissions or for advanced settings, click Advanced.

Windows XP access-control list management.

UNIT-VII

MAY-2011

6. (a) Discuss the file accessing methods. [8+8]
(b) Explain the techniques used to improve the efficiency and performance of secondary storage.

6. (a) Discuss the schemes for defining logical structure of a directory.
(b) Explain the different methods of maintaining free space list. [8+8]

6. (a) Explain the concept of file mounting.
(b) Explain the following methods of allocating disk space
1. Linked allocation
2. Indexed allocation [8+8]

6. (a) Explain the different file access control methods.
(b) Explain about virtual file systems. [8+8]

MAY-2010

6. What are the different file allocation methods? [16]

6. What are the different schemes for logical structure of a directory? [16]

6. (a) How locks are used in the context of files.
(b) Compare sequential and direct access methods. [5+11]

6. (a) What are the different file attributes and file operations.
(b) What are the different types of files supported by an operating system? [8+8]

NOV-2012

7. (a) Discuss in detail free space management .Also discuss about keeping track of free disk space in detail.
(b) How protection can be provided for file system?

7. (a) Explain in detail about the file attributes ,file operations and about the structure of a file system?
(b) What problems could occur if a system allowed a file system to be mounted simultaneously at more than one location?

7. (a) Explain the directory implementation of a file system in detail.
(b) What are the allocation methods of a file system? Explain briefly the indexed and linked allocation.

7. (a) Describe briefly the procedure of protecting files in the system.
(b) Explain in detail about the free space management in file system.

MAY-2012

7. a) Discuss the performance issues of tertiary-storage.
b) Explain the following techniques for structuring the page table in detail.
i) Hierarchical paging ii) Inverted page table [8+8]

7.b) Explain the following methods of allocating disk space
i) Linked allocation ii) Indexed allocation [8+8]

MAY-2010

7. What are the different types of mass storage structures? [16]

7. What are the different disk scheduling algorithms? [16]

MAY-2010

7 a) How access matrix can be used for providing protection. [16]

UNIT-VIII

MAY-2010

7. (a) How stable storage is implemented.

(b) With the help of a diagram explain the Disk structure. [8+8]

NOV-2012

8. (a) None of the disk-scheduling disciplines, except FCFS, is truly fair(Starvation may occur).
i) Explain why this assertion is true.
ii) Describe a way to modify algorithm such as SCAN to ensure fairness.
(b) Describe about disk attachment in detail.

8. (a) Elucidate disk structure in detail. Explain about disk scheduling in detail.
(b) What is disk scheduling? Explain in detail about FCFS and SSTF scheduling.

8. (a) Discuss in detail about variety of techniques to improve the efficiency and performance of secondary storage.
(b) Explain in detail about swap space management.

8. (a) Discuss briefly the general overview of the physical structure of secondary and tertiary storage devices.
(b) Explain in detail the two ways of disk storage in which the computers access.

MAY-2011

7. What are the different types of disk scheduling? Explain each with example. [16]

7. (a) Suppose that a disk drive has 200 cylinders numbered 0 to 199. The drive is currently serving a request at cylinder 95, The queue of pending requests, in FIFO order is
98, 181, 37,120,14,122,65,69
Starting from the current head position, What is the total distance (in cylinders) that the disk arm moves to satisfy all the pending requests for each of the following disk scheduling algorithm?
i) SCAN ii) C-SCAN
(b) Discuss about swap-space management. [8+8]

7. Discuss the following disk-scheduling algorithm with example
i. FCFS ii. SSTF iii. SCAN iv C-SCAN [16]

Introduction

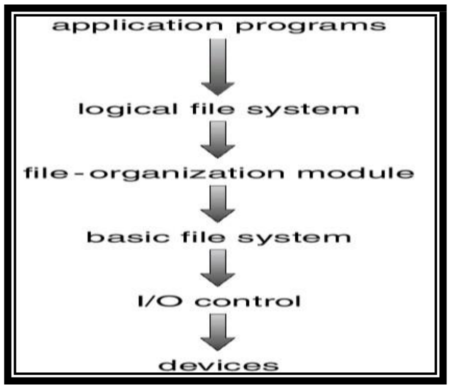
- The file system resides permanently on *secondary storage,* which is designed to hold a large amount of data permanently.
- This chapter is primarily concerned with issues surrounding file storage and access on the most common secondary-storage medium, the disk.

File-System Structure

- Disks provide the bulk of secondary storage on which a file system is maintained. They have two characteristics that make them a convenient medium for storing multiple files:

1. A disk can be rewritten in place; it is possible to read a block from the disk, modify the block, and write it back into the same place.
2. A disk can access directly any given block of information it contains. Thus, it is simple to access any file either sequentially or randomly, and switching from one file to another requires only moving the read-write heads and waiting for the disk to rotate.

- Rather than transferring a byte at a time, to improve I/O efficiency, I/O transfers between memory and disk are performed in units of *blocks.*
- Each block has one or more sectors. Depending on the disk drive, sectors vary from 32 bytes to 4,096 bytes; usually, they are 512 bytes.
- To provide efficient and convenient access to the disk, the operating system imposes one or more file systems to allow the data to be stored, located, and retrieved easily.
- A file system poses two quite different design problems.
  1. The first problem is defining how the file system should look to the user. This task involves defining a file and its attributes, the operations allowed on a file, and the directory structure for organizing files.
  2. The second problem is creating algorithms and data structures to map the logical file system onto the physical secondary-storage devices.
- The file system itself is generally composed of many different levels.
- Each level in the design uses the features of lower levels to create new features for use by higher levels.



Layered File System

I/O Control

- The lowest level, the *I/O control,* consists of **device drivers** and interrupts handlers to transfer information between the main memory and the disk system.
- A device driver can be thought of as a translator. Its input consists of high-level commands such as "retrieve block 123."
- Its output consists of low-level, hardware-specific instructions that are used by the hardware controller, which interfaces the I/O device to the rest of the system.
- The device driver usually writes specific bit patterns to special locations in the I/O controller's memory to tell the controller which device location to act on and what actions to take

Basic File System

- The **basic file system** needs only to issue generic commands to the appropriate device driver to read and write physical blocks on the disk.
- Each physical block is identified by its numeric disk address (for example, drive 1, cylinder 73, track 2, sector 10).

File- Organization Module

- The file-organization module knows about files and their logical blocks, as well as physical blocks.
- By knowing the type of file allocation used and the location of the file, the file-organization module can translate logical block addresses to physical block addresses for the basic file system to transfer.
- Each file's logical blocks are numbered from 0 (or 1) through *N*.
- The file-organization module also includes the free-space manager, which tracks unallocated blocks and provides these blocks to the file-organization module when requested.

Logical file System

- Finally, the logical file system manages metadata information. Metadata includes all of the file-system structure except the actual *data* (or contents of the files).
- The logical file system manages the directory structure. It maintains file structure via file-control blocks.
- A **file-control block** (FCB) contains information about the file, including ownership, permissions, and location of the file contents.
- The logical file system is also responsible for protection and security.

File-System implementation

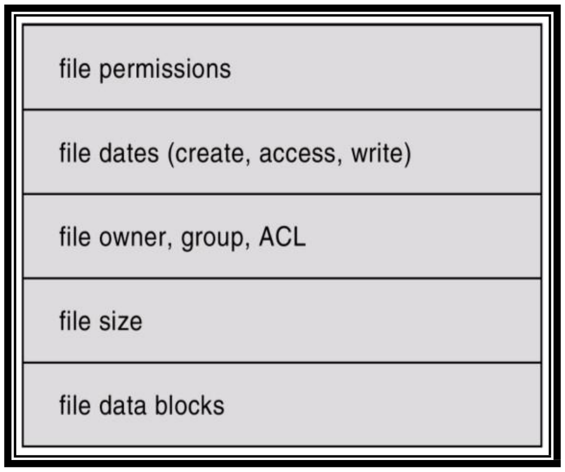
- The structures and operations used to implement file system operations.

Overview

- Several on-disk and in-memory structures are used to implement a file system.
- These structures vary depending on the operating system and the file system, but some general principles apply.
- On disk, the file system may contain information about how to boot an operating system stored there, the total number of blocks, the number and location of free blocks, the directory structure, and individual files.
- Many of these structures are detailed throughout the remainder of this chapter; here we describe them briefly:
  - **A boot control block** (per volume) can contain information needed by the system to boot an operating system from that volume. If the disk does not contain an operating system, this block can be empty. It is typically the first block of a volume.UFS, it is called the boot block; in NTFS, it is the partition boot **sector.**

- ❖ A **volume control block** (per volume) contains volume (or partition) details, such as the number of blocks in the partition, size of the blocks, free block count and free-block pointers, and free FCB count and FCB pointers. In UFS, this is called a **superblock;** in NTFS, it is stored in. the **master file table.**
- ❖ A **directory structure** per file system is used to organize the files. In UFS, this includes file names and associated **inode** numbers. In NTFS it is stored in the **master file table.**
- ❖ A **per-file FCB** contains many details about the file, including file permissions, ownership, size, and location of the data blocks. In UFS, this is called the inode. In NTFS, this information is actually stored within the master file table, which uses a relational database structure, with a row per file.

- ➢ The in-memory information is used for both file-system management and performance improvement via caching. The data are loaded at mount time and discarded at dismount. The structures may include the ones described below:
  - ❖ An **in-memory mount table** contains information about each mounted volume.
  - ❖ An **in-memory directory-structure** cache holds the directory information of recently accessed directories. (For directories at which volumes are mounted, it can contain a pointer to the volume table.)
  - ❖ The **system-wide open-file table** contains a copy of the FCB of each open file, as well as other information.
  - ❖ The **per-process open-file table** contains a pointer to the appropriate entry in the system-wide open-file table, as well as other information.
- ➢ To create a new file, an application program calls the logical file system.
- ➢ The logical file system knows the format of the directory structures.
- ➢ To create a new file, it allocates a new FCB. (Alternatively, if the file-system implementation creates all FCBs at file-system creation time, an FCB is allocated from the set of free FCBs.)
- ➢ The system then reads the appropriate directory into memory updates it with the new file name and FCB, and writes it back to the disk.

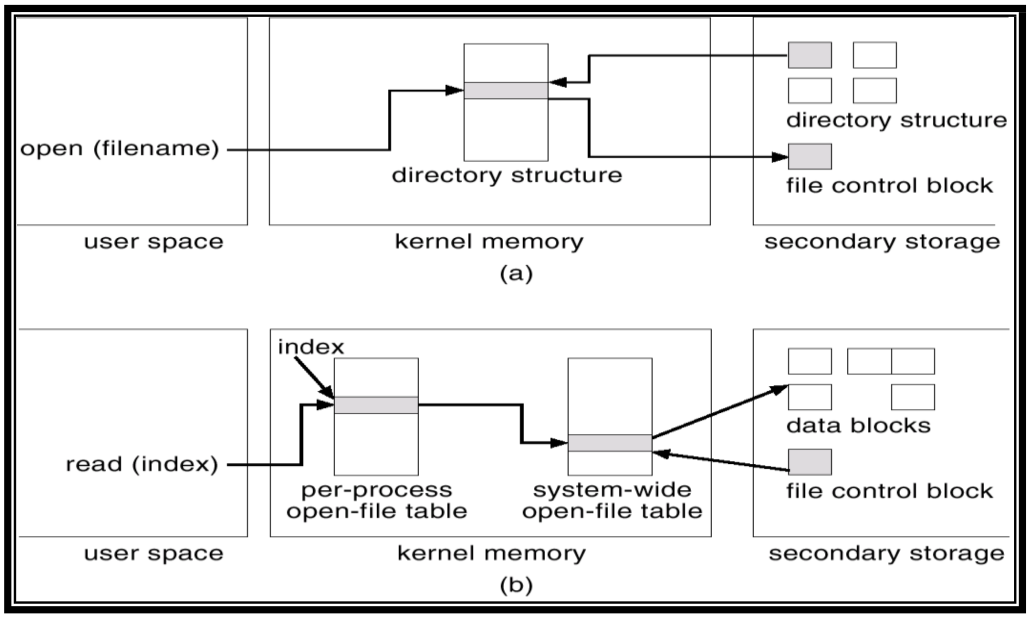


A Typical File Control Block

- ➢ The logical file system can call the file-organization module to map the directory I/O into disk-block numbers, which are passed on to the basic file system and I/O control system.
- ➢ Now that a file has been created, it can be used for I/O. First, though, it must be *opened.*

- The open() call passes a file name to the file system. The open() system call first searches the system-wide open-file table to see if the file is already in use by another process.
- If it is, a per-process open-file table entry is created pointing to the existing system-wide open-file table.
- This algorithm can save substantial overhead. When a file is opened, the directory structure is searched for the given file name.
- Parts of the directory structure are usually cached in memory to speed directory operations.
- Once the file is found, the FCB is copied into a system-wide open-file table in memory. This table not only stores the FCB but also tracks the number of processes that have the file open.
- Next, an entry is made in the per-process open-file table, with a pointer to the entry in the system-wide open-file table and some other fields.
- These other fields can include a pointer to the current location in the file (for the next read() or write () operation) and the access mode in which the file is open.
- The open() call returns a pointer to the appropriate entry in the per-process file-system table.
- All file operations are then performed via this pointer.
- UNIX systems refer to it as a file descriptor; Windows refers to it as a **file handle.** Consequently, as long as the file is not closed, all file operations are done on the open-file table.

- When a process closes the file, the per-process table entry is removed, and the system-wide entry's open count is decremented.
- When all users that have opened the file close it, any updated metadata is copied back to the disk-based directory structure, and the system-wide open-file table entry is removed.



In-Memory File System Structures (a) File open, (b) File read.

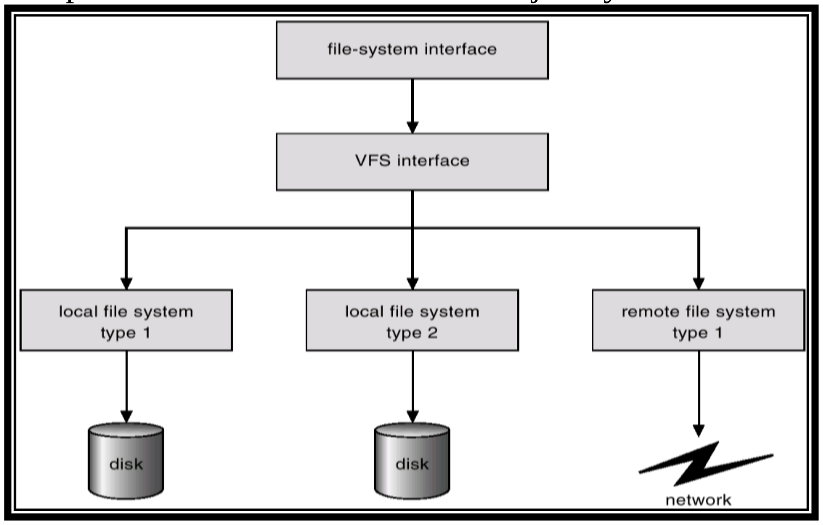
Partitions and Mounting

- The layout of a disk can have many variations, depending on the operating system.

- A disk can be sliced into multiple partitions, or a volume can span multiple partitions on multiple disks.
- Each partition can be either "raw," containing no file system, or "cooked;' containing a file system.
- Raw disk is used where no file system is appropriate. UNIX swap space can use a raw partition, for example, as it uses its own format on disk and does not use a file system.
- Boot information can be stored in a separate partition. Again, it has its own format, because at boot time the system does not have file-system device drivers loaded and therefore cannot interpret the file-system format.
- Rather, boot information is usually a sequential series of blocks, loaded as an image into memory.
- Execution of the image starts at a predefined location, such as the first byte.
- This boot image can contain more than the instructions for how to boot a specific operating system.
- Multiple operating systems can be installed, on such a system. How does the system know which one to boot? A boot loader that understands multiple file systems and multiple operating systems can occupy the boot space.
- Once loaded, it can boot one of the operating systems available on the disk. The disk can have multiple partitions, each containing a different type of file system and a different operating system.
- The root **partition,** which contains the operating-system kernel and sometimes other system files, is mounted at boot time.
- Other volumes can be automatically mounted at boot or manually mounted later, depending on the operating system.
- As part of a successful mount operation, the operating system verifies that the device contains a valid file system.
- It does so by asking the device driver to read the device directory and verifying that the directory has the expected format.
- If the format is invalid, the partition must have its consistency checked and possibly corrected, either with or without user intervention.
- Finally, the operating system notes in its in-memory **mount table** structure that a file system is mounted, along with the type of the file system.
- Microsoft Windows-based systems mount each volume in a separate name space, denoted by a letter and a colon.
- To record that a file system is mounted at F:, for example, the operating system places a pointer to the file system in a field of the device structure corresponding to F:.
- On UNIX, file systems can be mounted at any directory. Mounting is implemented by setting a flag in the in-memory copy of the inode for that directory.
- The flag indicates that the directory is a mount point. A field then points to an entry in the mount table, indicating which device is mounted there.
- The mount table entry contains a pointer to the superblock of the file system on that device.
- This scheme enables the operating system to traverse its directory structure, switching among file systems of varying types, seamlessly.

Virtual File Systems

- The previous section makes it clear that modern operating systems must concurrently support multiple types of file systems. But how does an operating system allow multiple types of file systems to be integrated into a directory structure?
- An obvious but suboptimal method of implementing multiple types of file systems is to write directory and file routines for each type.
- Most operating systems, including UNIX, use object-oriented techniques to simplify, organize, and modularize the implementation.
- The use of these methods allows very dissimilar file-system types to be implemented within the same structure, including network file systems, such as NFS.
- Users can access files that are contained within multiple file systems on the local disk or even on file systems available across the network.
- Data structures and procedures are used to isolate the basic system call functionality from the implementation details.
- Thus, the file-system implementation consists of three major layers.



Schematic View of Virtual File System

- The first layer is the file-system interface, based on the open(),read(), write (), and close () calls and on file descriptors.
- The second layer is called the virtual file system (VFS) layer; it serves two important functions:

  1. It separates file-system-generic operations from their implementation by defining a clean VFS interface.

  2. The VFS provides a mechanism for uniquely representing a file throughout a network. The VFS is based on a file-representation structure, called a vnode, that contains a numerical designator for a network-wide unique file. (UNIX inodes are unique within only a single file system.) .The kernel maintains one vnode structure for each active node (file or directory).
- The VFS distinguishes local files from remote ones, and local files arefurther distinguished according to their file-system types.
- The VFS activates file-system-specific operations to handle local requests according to their file-system types and even calls the NFS protocol procedures for remote requests.

- File handles are constructed from the relevant vnodes and are passed as arguments to these procedures. The layer implementing the file system type or the remote-file-system protocol is the third layer of the architecture.
- Let's briefly examine the VFS architecture in Linux. The four main object types defined by the Linux VFS are:
  - The **inode object,** which represents an individual file
  - The **file object,** which represents an open file
  - The **superblock object,** which represents an entire file system
  - The **dentry object,** which represents an individual directory entry
- For each of these four object types, the VFS defines a set of operations that must be implemented.
- Every object of one of these types contains a pointer to a function table.
- **The** function table lists the addresses of the actual functions that implement the defined operations for that particular object.
- For example, an abbreviated API for some of the operations for the file object include:
  - int open(. . .) —Open a file.
  - ssize_t read(. . .)—Read from a file.
  - ssize_t write (. . .) —Write to a file.
  - Int mmap(. . .) — Memory-map a file.
- An implementation of the file object for a specific file type is required to implement each function specified in the definition of the file object.

<u>Directory implementation</u>

- The selection of directory-allocation and directory-management algorithms
- significantly affects the efficiency, performance, and reliability of the file
- system. In this section, we discuss the trade-offs involved in choosing one
- of these algorithms.

<u>11.3.1 Linear List</u>

- The simplest method of implementing a directory is to use a linear list of file names with pointers to the data blocks.
- This method is simple to program but time-consuming to execute.
- To create a new file. We must first search the directory to be sure that no existing file has the same name.
- Then, we add a new entry at the end of the directory.
- To delete a file, we search the directory for the named file, then release the space allocated to it.
- To reuse the directory entry, we can do one of several things. We can mark the entry as unused (by assigning it a special name, such as an all-blank name, or with a used-unused, bit in each entry), or we can attach it to a list of free directory entries.
- A third alternative is to copy the last entry in the directory into the freed location and to decrease the length of the directory. A linked list can also be used to decrease the time required to delete a file.

- The real disadvantage of a linear list of directory entries is that finding a file requires a linear search.
- Directory information is used frequently, and users will notice if access to it is slow.
- In fact, many operating systems implement a software cache to store the most recently used directory information.
- A cache hit avoids the need to constantly reread the information from disk.
- A sorted list allows a binary search and decreases the average search time. However, the requirement that the list be kept sorted may complicate creating and deleting files, since we may have to move substantial amounts of directory information to maintain a sorted directory.
- A more sophisticated tree data structure, such as a B-tree, might help here. An advantage of the sorted list is that a sorted directory listing can be produced without a separate sort step.

7.3.2 Hash Table

- Another data structure used for a file directory is a **hash table.** With this method, a linear list stores the directory entries, but a hash data structure is also used.
- The hash table takes a value computed from the file name and returns a pointer to the file name in the linear list.
  - Therefore, it can greatly decrease the directory search time. Insertion and deletion are also fairly straightforward, although some provision must be made for **collisions**—situations in which two file names hash to the same location.
- For example, assume that we make a linear-probing hash table that holds 64 entries. The hash function converts file names into integers from 0 to 63, for instance, by using the remainder of a division by 64.
- If we later try to create a 65th file, we must enlarge the directory hash table—say, to 128 entries.
- As a result, we need a new hash function that must map file names to the range 0 to 127, and we must reorganize the existing directory entries to reflect their new hash-function values.
- Alternatively, a chained-overflow hash table can be used. Each hash entry can be a linked list instead of an individual value, and we can resolve collisions by adding the new entry to the linked list.
- Lookups may be somewhat slowed, because searching for a name might require stepping through a linked list of colliding table entries.
- Still, this method is likely to be much faster than a linear search through the entire directory.

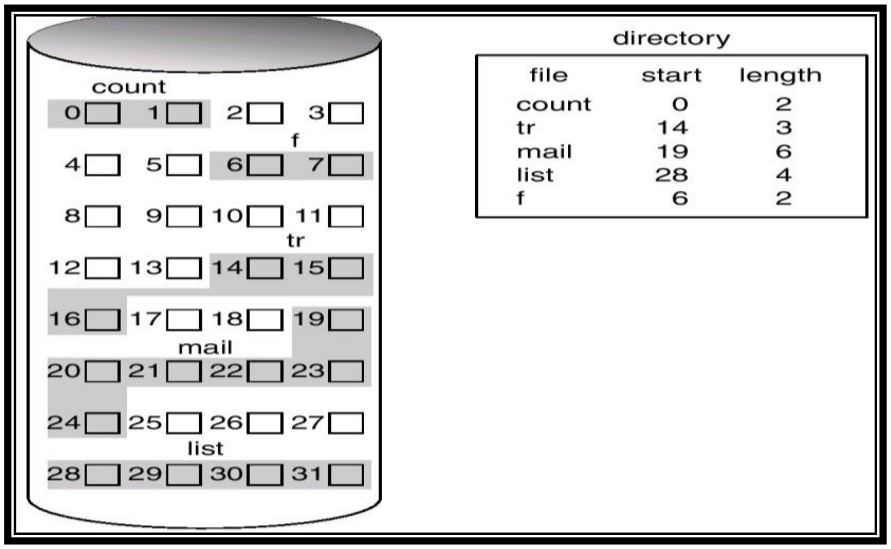
Allocation Methods

- The direct-access nature of disks allows us flexibility in the implementation of files, in almost every case, many files are stored on the same disk.
- The main problem is how to allocate space to these files so that disk space is utilized effectively and files can be accessed quickly.
- Three major methods of allocating disk space are in wide use:
  1. Contiguous
  2. Linked
  3. indexed.

- Some systems (such as Data General's RDOS for its Nova line of computers) support all three. More commonly, a system uses one method for all files within a file system type.

7.4.1 Contiguous Allocation

- Contiguous allocation requires that each file occupy a set of contiguous blocks on the disk.
- Disk addresses define a linear ordering on the disk.
- With this ordering, assuming that only one job is accessing the disk, accessing block *b* +1 after block *b* normally requires no head movement.
- When head movement is needed (from the last sector of one cylinder to the first sector of the next cylinder), the head need only move from one track to the next.
- Thus, the number of disk seeks required for accessing contiguously allocated files is minimal, as is seek time when a seek is finally needed.
- Contiguous allocation of a file is defined by the disk address and length (in block units) of the first block.
- If the file is *n* blocks long and starts at location *b,* then it occupies blocks *b, b* + 1, *b + 2, ..., b + n* -1.
- The directory entry for each file indicates the address of the starting block and the length of the area allocated for this file.



file	start	length
count	0	2
tr	14	3
mail	19	6
list	28	4
f	6	2

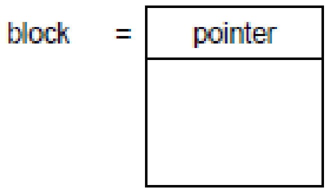
Contiguous Allocation of Disk Space

- Accessing a file that has been allocated contiguously is easy. For sequential access, the file system remembers the disk address of the last block referenced and, when necessary, reads the next block.
- For direct access to block 'i' of a file that starts at block *b,* we can immediately access block *b* + *i.*
- Thus, both sequential and direct access can be supported by contiguous allocation.
- Contiguous allocation has some problems, however. One difficulty is finding space for a new file,which involves how to satisfy a request of size *n* from a list of free holes.
- First fit and best fit are the most common strategies used to select a free hole from the set of available holes.
- All these algorithms suffer from the problem of **external fragmentation.**

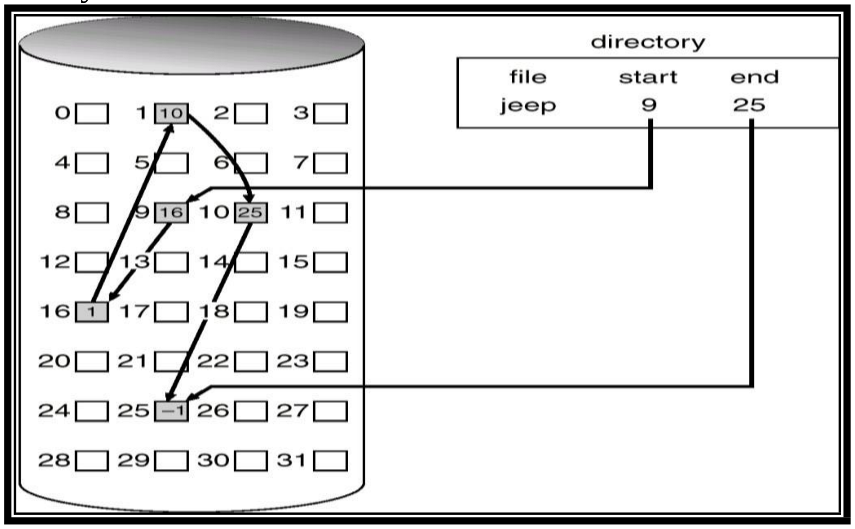
- Another problem with contiguous allocation is determining how much space is needed for a file.
- When the file is created, the total amount of space it will need must be found and allocated.
- If we allocate too little space to a file, we may find that the file cannot be extended.
- Two possibilities then, exist. First, the user program can be terminated with an appropriate error message. The user must then allocate more space and run the program again.
- The other possibility is to find a larger hole, copy the contents of the file to the new space, and release the previous space.
- Even if the total amount of space needed for a file is known in advance, pre allocation may be inefficient.
- To minimize these drawbacks, some operating systems use a modified contiguous-allocation scheme.
- Extent-based file systems allocate disk blocks in extents.
- An extent is a contiguous block of disks. Extents are allocated for file allocation. A file consists of one or more extents.

11.4.2 Linked Allocation

- **Linked allocation** solves all problems of contiguous allocation.
- With linked allocation, each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk.
- Each block contains a pointer to the next block. These pointers are not made available to the user.



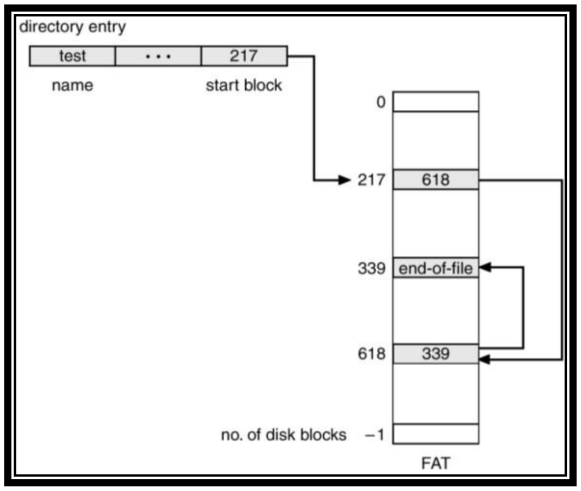
- The directory contains a pointer to the first and last blocks of the file.
- For example, a file of five blocks might start at block 9 and continue at block 16, then block 1, then block 10, and finally block 25 .



Linked Allocation

- Thus, if each block is 512 bytes in size, and a disk address (the pointer) requires 4 bytes, then the user sees blocks of 508 bytes.
- To create a new file, we simply create a new entry in the directory.
- With linked allocation, each directory entry has a pointer to the first disk block of the file.
- This pointer is initialized to *nil* (the end-of-list pointer value) to signify an empty file.
- The size field is also set to 0. A write to the file causes the free-space management system to find a free block, and this new block is written to and is linked to the end of the file.
- To read a file, we simply read blocks by following the pointers from block to block.
- There is no external fragmentation with linked allocation, and any free block on the free-space list can be used to satisfy a request.
- The size of a file need not be declared when that file is created.
- A file can continue to grow as long as free blocks are available. Consequently, it is never necessary to compact disk space.
- Linked allocation does have disadvantages, however. The major problem is that it can be used effectively only for sequential-access files.
- To find the ith block of a file, we must start at the beginning of that file and follow the pointers until we get to the ith block.
- Each access to a pointer requires a disk read, and some require a disk seek.
- Another disadvantage is the space required for the pointers. If a pointer requires 4 bytes out of a 512-byte block, then 0.78 percent of the disk is being used for pointers, rather than for information.
- The usual solution to this problem is to collect blocks into multiples, called clusters, and to allocate clusters rather than blocks.
- For instance, the file system may define a cluster as four blocks and operate on the disk only in cluster units.
- Pointers then use a much smaller percentage of the file's disk space.
- Yet another problem of linked allocation is reliability. Recall that the files are linked together by pointers scattered all over the disk, and consider what would happen if a pointer were lost or damaged.
- An important variation on linked allocation is the use of a **file-allocation table** (FAT). This simple but efficient method of disk-space allocation is used by the MS-DOS and OS/2 operating systems.
- A section of disk at the beginning of each volume is set aside to contain the table.
- The table has one entry for each disk block and is indexed by block number.
- The FAT is used in much the same way as a linked list. The directory entry contains the block number of the first block of the file.
- The table entry indexed by that block number contains the block number of the next block in the file.
- This chain continues until the last block, which has a special end-of-file value as the table entry.
- Unused blocks are indicated by a 0 table value.
- Allocating a new block to a file is a simple matter of finding the first 0-valued table entry and replacing the previous end-of-file value with the address of the new block.
- The 0 is then replaced with the end-of-file value.
- An illustrative example is the FAT structure shown in below for a file consisting of disk blocks 217, 618, and 339.

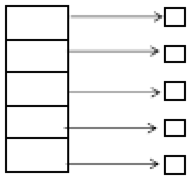
- The FAT allocation scheme can result in a significant number of disk head seeks, unless the FAT is cached.



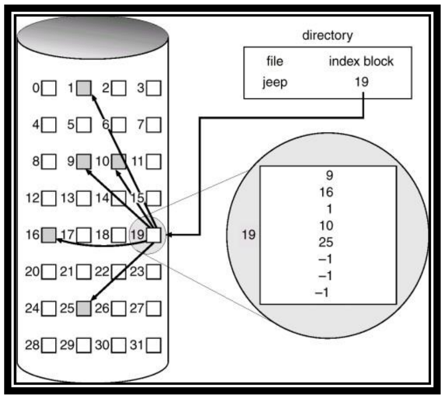
File-Allocation Table

7.4.3 Indexed Allocation

- Linked allocation solves the external-fragmentation and size-declaration problems of contiguous allocation.
- However, in the absence of a FAT, linked allocation cannot support efficient direct access, since the pointers to the blocks are scattered with the blocks themselves all over the disk and must be retrieved in order.
- **Indexed allocation** solves this problem by bringing all the pointers together into one location: the **index block.**



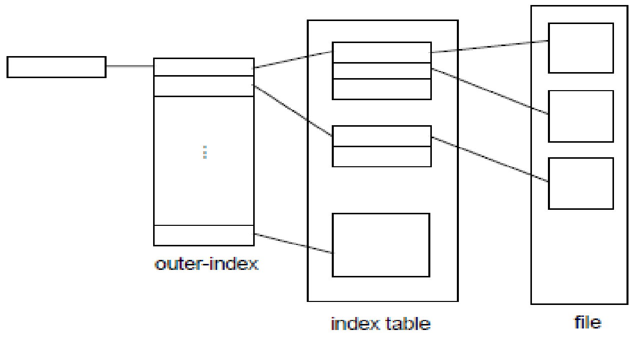
- Each file has its own index block, which is an array of disk-block addresses.
- The ith entry in the index block points to the ith block of the file.
- The directory contains the address of the index block . To find and read the ith block, we use the pointer in the ith index-block entry.



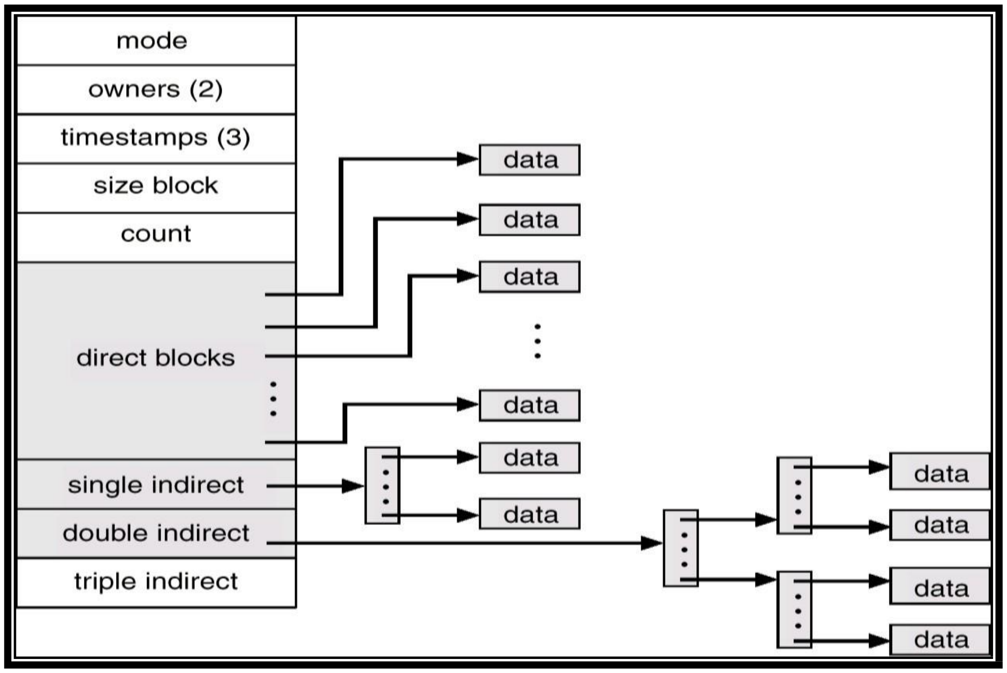
Example of Indexed Allocation

- When the file is created, all pointers in the index block are set to *nil.*
- When the ith block is first written, a block is obtained from the free-space manager, and its address is put in the ith index-block entry.
- Indexed allocation supports direct access, without suffering from external fragmentation, because any free block on the disk can satisfy a request for more space.
- Indexed allocation does suffer from wasted space, however. The pointer overhead of the index block is generally greater than the pointer overhead of linked allocation.
- Consider a common case in which we have a file of only one or two blocks. With linked allocation, we lose the space of only one pointer per block.
- With indexed allocation, an entire index block must be allocated, even if only one or two pointers will be *non-nil.*
- This point raises the question of how large the index block should be. Every file must have an index block, so we want the index block to be as small as possible.
- If the index block is too small, however, it will not be able to hold enough pointers for a large file, and a mechanism will have to be available to deal with this issue.
- Mechanisms for this purpose include the following:
  - **Linked scheme**. An index block is normally one disk block. Thus, it can be read and written directly by itself. To allow for large files, we can link together several index blocks. For example, an index block might contain a small header giving the name of the file and a set of the first 100 disk-block addresses. The next address (the last word in the index block) is *nil* (for a small file) or is a pointer to another index block (for a large file).

  - **Multilevel index**. A variant of the linked representation is to use a first level index block to point to a set of second-level index blocks, which in turn point to the file blocks. To access a block, the operating system uses the first-level index to find a second-level index block and then uses that block to find the desired data block. This approach could be continued to a third or fourth level, depending on the desired maximum file size. With 4,096-byte blocks, we could store 1,024 4-byte pointers in an index block. Two levels of indexes allow 1,048,576 data blocks and a file size of up to 4 GB.



- ❖ **Combined scheme.** Another alternative, used in the UFS, is to keep the first, say, 15 pointers of the index block in the file's inode.
  - ➢ The first 12 of these pointers point to **direct blocks;** that is, they contain addresses of blocks that contain data of the file. Thus, the data for small files (of no more than 12 blocks) do not need a separate index block. If the block size is 4 KB, then up to 48 KB of data can be accessed directly.
  - ➢ The next three pointers point to **indirect blocks.**
  - ➢ The first points to a **single indirect block,** which is an index block containing not data but the addresses of blocks that do contain data.
  - ➢ The second points to a **double indirect block,** which contains the address of a block that contains the addresses of blocks that contain pointers to the actual data blocks.
  - ➢ The last pointer contains the address of a **triple indirect block.**
  - ➢ Under this method, the number of blocks that can be allocated to a file exceeds the amount of space addressable by the 4-byte file pointers used by many operating systems. A 32-bit file pointer reaches only 232bytes, or 4 GB. Many UNIX implementations, including Solaris and IBM's A1X, now support up to 64-bit file pointers. Pointers of this size allow files and file systems to be terabytes in size. A UNIX inode is shown in below.



Combined Scheme: UNIX (4K bytes per block)

- Indexed-allocation schemes suffer from some of the same performance problems as does linked allocation. Specifically, the index blocks can be cached in memory, but the data blocks may be spread all over a volume.

Free-Space Management

- Since disk space is limited, we need to reuse the space from deleted files for new files, if possible.
- To keep track of free disk space, the system maintains a **free-space list.**
- The free-space list records *all free disk* blocks—those not allocated to some file or directory.
- To create a file, we search the free-space list for the required amount of space and allocate that space to the new file. This space is then removed from the free-space list.
- When a file is deleted, its disk space is added to the free-space list.
- Free space can be implemented in many ways:
    1. Bit Vector
    2. Linked List
    3. Grouping
    4. Counting

Bit Vector

- Frequently, the free-space list is implemented as a bit **map** or bit vector. Each block is represented by 1 bit.
- If the block is free, the bit is 1; if the block is allocated, the bit is 0.

- For example, consider a disk where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25,26, and 27 are free and the rest of the blocks are allocated.
- The free-space bit map would be

  001111001111110001100000011100000...

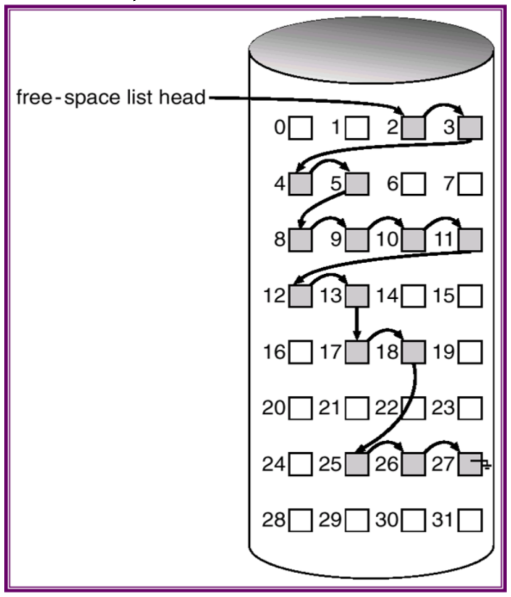
- The main advantage of this approach is its relative simplicity and its efficiency in finding the first free block or *n* consecutive free blocks on the disk, indeed, many computers supply bit-manipulation instructions that can be used effectively for that purpose.
- One technique for finding the first free block on a system that uses a bit-vector to allocate disk space is to sequentially check each word in the bit map to see whether that value is not 0, since a 0-valued word has all 0 bits and represents a set of allocated blocks.
- The first non-0 word is scanned for the first 1 bit, which is the location of the first free block.
- The calculation of the block number is

  (number of bits per word) x (number of 0-value words) + offset of first 1 bit.

- Bit vectors are inefficient unless the entire vector is kept in main memory.
- Keeping it in main memory is possible for smaller disks but not necessarily for larger ones.

Linked List

- Another approach to free-space management is to link together all the free disk blocks, keeping a pointer to the first free block in a special location on the disk and caching it in memory.
- This first block contains a pointer to the next free disk block, and so on.
- In our earlier example, we would keep a pointer to block 2 as the first free block.
- Block *2* would contain a pointer to block 3, which would point to block 4, which would point to block 5, which would point to block 8, and so on shown in below.



Linked Free Space List on Disk

- However; this scheme is not efficient; to traverse the list, we must read each block, which requires substantial I/O time. Fortunately, traversing the free list is not a frequent action. Usually, the

Grouping

- A modification of the free-list approach is to store the addresses of *n* free blocks in the first free block. The first n—1 of these blocks is actually free.
- The last block contains the addresses of other n free blocks, and so on.
- The addresses of a large number of free blocks can now be found quickly, unlike the situation
- when the standard linked-list approach is used.

Counting

- Another approach is to take advantage of the fact that, generally, several contiguous blocks may be allocated or freed simultaneously, particularly when space is allocated with the contiguous-allocation algorithm or through clustering.
- Thus, rather than keeping a list of *n* free disk addresses, we can keep the address of the first free block and the number *n* of free contiguous blocks that follow the first block.
- Each entry in the free-space list then consists of a disk address and a count.
- Although each entry requires more space than would a simple disk address, the overall list will be shorter, as long as the count is generally greater than 1.

1. a) Explain the file access methods.
 b) Describe about Directory Structure &its types.

2. a) What is file mounting & Mounting Point ? Explain the necessisity of file mounting while we accessing a file.
b) How file be shared &how they are protected.

3. a)Describe about file system structure & how it can be implemented.
 b) Discuss the techniques to implement the Directory.

4. Explain the file allocation methods

5. Describe the necessisity of free space management.

1 a) Explain Disk Structure with neat Diagram.
 b) What is Disk attachment? Explain the ways of Disk attachment.

2. Explain following Disk Scheduling algorithms with example

a) FCFS b) SSTF c) SCAN d) C-SCAN e)LOOK f) C-LOOK

(OR)

Suppose that a disk drive has 5,000 cylinders, numbered 0 to 4999. The drive is currently serving a request at cylinder 143, and the previous request was at cylinder 125. The queue of pending requests, in FIFO order, is:

86, 1470, 913, 1774, 948, 1509, 1022, 1750,130

Starting from the current head position, what is the total distance (in cylinders) that the disk arm moves to satisfy all the pending requests for each of the following disk-scheduling algorithms?

a. FCFS
b. SSTF
c. SCAN
d. LOOK
e. C-SCAN
f. C-LOOK

3. Explain Swap-Space management.