Deadlocks

- A set of process is in a deadlock state if each process in the set is waiting for an event that can be caused by only another process in the set.
- In other words, each member of the set of deadlock processes is waiting for a resource that can be released only by a deadlock process.
- None of the processes can run, none of them can release any resources, and none of them can be awakened.
- The resources may be either physical or logical.
- Examples of physical resources are Printers, Hard Disc Drives, Memory Space, and CPU Cycles.
- Examples of logical resources are Files, Semaphores, and Monitors.
- The simplest example of deadlock is where process 1 has been allocated non-shareable resources A (say a Hard Disc drive) and process 2 has be allocated non-sharable resource B (say a printer). Now, if it turns
- out that process 1 needs resource B (printer) to proceed and process 2 needs resource A (Hard Disc drive) to proceed and these are the only two processes in the system, each is blocked the other and all useful work in the system stops. This situation is termed deadlock. The system is in deadlock state because each process holds a resource being requested by the other process neither process is willing to release the resource it holds.

Preemptable and Nonpreemptable Resources

- Resources come in two flavors: preemptable and nonpreemptable.
- A preemptable resource is one that can be taken away from the process with no ill effects. Memory is an example of a preemptable resource.
- A nonpreemptable resource is one that cannot be taken away from process (without causing ill effect).For example, CD resources are not preemptable at an arbitrary moment.
- Reallocating resources can resolve deadlocks that involve preemptable resources. Deadlocks that involve nonpreemptable resources are difficult to deal with.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, then the requesting process must wait until it can acquire the resource.
2. **Use:** The process can operate on the resource.
3. **Release:** The process releases the resource.

Necessary Conditions for Deadlock

Coffman (1971) identified four conditions that must hold simultaneously for there to be a deadlock.

1. Mutual Exclusion Condition: The resources involved are non-shareable.

Explanation: At least one resource must be held in a non-shareable mode, that is, only one process at a time claims exclusive control of the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

2. Hold and Wait Condition: Requesting process hold already the resources while waiting for requested resources.

Explanation: There must exist a process that is holding a resource already allocated to it while waiting for additional resource that are currently being held by other processes.

4. **No-Preemptive Condition:** Resources already allocated to a process cannot be preempted.

Explanation: Resources cannot be removed from the processes are used to completion or released voluntarily by the process holding it.

4. Circular Wait Condition: The processes in the system form a circular list or chain where each process in the list is waiting for a resource held by the next process in the list. There exists a set {P0, P1, …, P0} of waiting processes such that P0 is waiting for a resource that is held by P1, P1 is waiting for a resource that is held by P2, …, Pn–1 is waiting for a resource that is held by Pn, and P0 is waiting for a resource that is held by P0.

Note: It is not possible to have a deadlock involving only one single process. The deadlock involves a circular "hold-and-wait" condition between two or more processes, so "one" process cannot hold a resource, yet be waiting for another resource that it is holding. In addition, deadlock is not possible between two threads in a process, because it is the process that holds resources, not the thread that is, each thread has access to the resources held by the process.

Resource-Allocation Graph

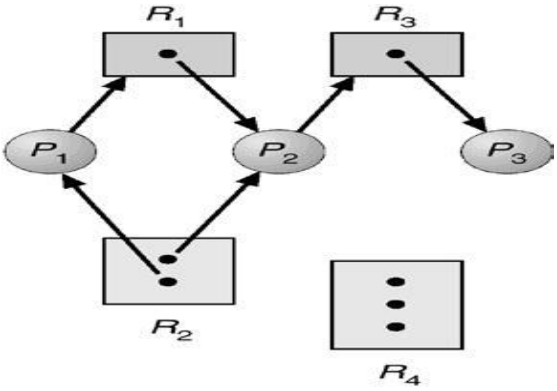
Deadlocks can be described in terms of a directed graph called a **system resource-allocation graph**.
This graph consists of a set of vertices V and a set of edges E.
The set of vertices V is partitioned into two different types of nodes P = {PI, P2, ..., Pn}, the set consisting of all the active processes 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 P->Rj;
it signifies that process Pi requested an instance of resource type Rj and is currently waiting for that resource.
A directed edge Pi ->Rj is called a request edge.
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 Rj->Pi is called an assignment edge.
Pictorially, we represent each process Pi as a circle and each resource type Rj as a square. Since resource type Rj may have more than one instance, we represent each such instance as a dot within the square.

The resource-allocation graph shown below depicts the following situation.
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:

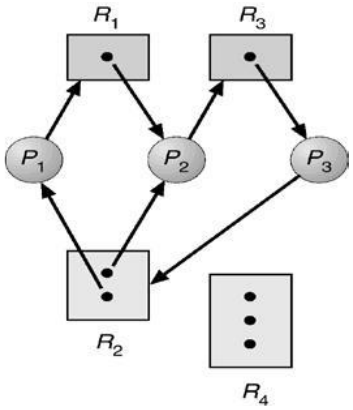
- Process PI is holding an instance of resource type R2, and is waiting for an instance of resource type R1
- Process P2 is holding an instance of R1 and R2, and is waiting for an instance of resource type R3.
- Process P3 is holding an instance of R3.



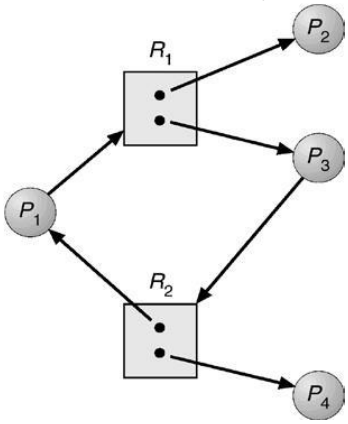
Given the definition of a resource-allocation graph, it can be shown that,

- 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. In this case, a cycle in the graph is both a necessary
- and a sufficient condition for the existence of deadlock.
- If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has
- occurred.
- In this case, a cycle in the graph is a necessary but not a sufficient condition for the existence of
- deadlock.

To illustrate this concept, let us return to the resource-allocation graph depicted in Figure.
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
Processes PI, P2, and P3 are deadlocked. Process P2 is waiting for the resource R3, which is held by process P3. Process P3, on the other hand, is waiting for either process PI or process P2 to release resource R2. In addition, process PI is waiting for process P2 to release resource R1.



Now consider the resource-allocation graph in the following Figure. In this example, we also have a cycle However, there is no deadlock. Observe that process P4 may release its instance of resource type R2. That resource can then be allocated to P3, breaking the cycle.



METHODS OF HANDLING DEADLOCK

In general, there are four strategies of dealing with deadlock problem:

1. **Deadlock Prevention**: Prevent deadlock by resource scheduling so as to negate at least one of the four conditions.
2. **Deadlock Avoidance**: Avoid deadlock by careful resource scheduling.
3. **Deadlock Detection and Recovery**: Detect deadlock and when it occurs, take steps to recover.
4. **The Ostrich Approach:** Just ignore the deadlock problem altogether.

DEADLOCK PREVENTION

A deadlock may be prevented by denying any one of the conditions.

• **Elimination of "Mutual Exclusion" Condition:** The mutual exclusion condition must hold for non-sharable resources. That is, several processes cannot simultaneously share a single resource. This condition is difficult to eliminate because some resources, such as the Hard disc drive and printer, are inherently non-shareable. Note that shareable resources like read-only-file do not require mutually exclusive access and thus cannot be involved in deadlock.

• **Elimination of "Hold and Wait" Condition:** There are two possibilities for elimination of the second condition.

1. The first alternative is that a process request be granted all of the resources it needs at once, prior to execution.

2. The second alternative is to disallow a process from requesting resources whenever it has previously allocated resources.

This strategy requires that all of the resources a process will need must be requested at once. The system must grant resources on "all or none" basis. If the complete set of resources needed by a process is not currently available, then the process must wait until the complete set is available. While the process waits, however, it may not hold any resources. Thus the "wait for" condition is denied and deadlocks cannot occur. This strategy can lead to serious waste of resources.

• **Elimination of "No-preemption" Condition:** The non-preemption condition can be alleviated by forcing a process waiting for a resource that cannot immediately be allocated to relinquish all of its currently held resources, so that other processes may use them to finish. This strategy requires that when a process that is holding some resources is denied a request for additional resources. The process must release its held resources and, if necessary, request them again together with additional resources. Implementation of this strategy denies the "no-preemptive" condition effectively.

• **Elimination of "Circular Wait" Condition:** The last condition, the circular wait, can be denied by imposing a total ordering on all of the resource types and then forcing, all processes to request the resources in order (increasing or decreasing). This strategy impose a total ordering of all resources types, and to require that each process requests resources in a numerical order (increasing or decreasing) of enumeration. With this rule, the resource allocation graph can never have a cycle.

For example, provide a global numbering of all the resources, as shown
1 ≡ Card reader
2 ≡ Printer
3 ≡ Optical driver
4 ≡ HDD
5 ≡ Card punch
Now the rule is this: processes can request resources whenever they want to, but all requests must be made in numerical order. A process may request first printer and then a HDD(order: 2, 4), but it may not request first a optical driver and then a printer (order: 3, 2). The problem with this strategy is that it may be impossible to find an ordering that satisfies everyone.

DEADLOCK AVOIDANCE

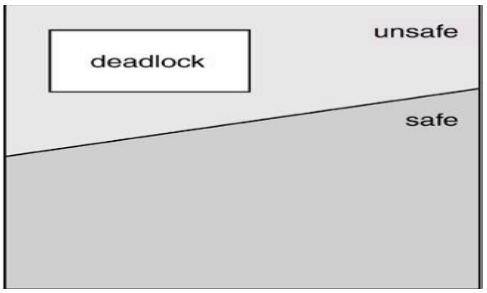
This approach to the deadlock problem anticipates deadlock before it actually occurs. This approach employs an algorithm to access the possibility that deadlock could occur and acting accordingly. If the necessary conditions for a deadlock are in place, it is still possible to avoid deadlock by being careful when resources are allocated. It employs the most famous deadlock avoidance algorithm that is the Banker's algorithm.

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.

Safe and Unsafe States

A system is said to be in a **Safe State**, if there is a safe execution sequence. An execution sequence is an ordering for process execution such that each process runs until it terminates or blocked and all request for resources are immediately granted if the resource is available.

A system is said to be in an **Unsafe State**, if there is no safe execution sequence. An unsafe state may not be deadlocked, but there is at least one sequence of requests from processes that would make the system deadlocked.



Resource-Allocation Graph Algorithm

The deadlock avoidance algorithm uses a variant of the resource-allocation graph to avoid deadlocked state.

- It introduces 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 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.

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 that does not result in the formation of a cycle in the resource-allocation graph.

An algorithm for detecting a cycle in this graph is called cycle detection algorithm. 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.

Banker's algorithm

- The **Banker's algorithm** is a resource allocation & deadlock avoidance algorithm developed by Edsger Dijkstra that test for safety by simulating the allocation of pre-determined maximum possible amounts of all resources. Then it makes a "safe-state" check to test for possible deadlock conditions for all other pending activities, before deciding whether allocation should be allowed to continue.
- The Banker's algorithm is run by the operating system whenever a process requests resources. The algorithm prevents deadlock by denying or postponing the request if it determines that accepting the request could put the system in an unsafe state (one where deadlock could occur).

For the Banker's algorithm to work, it needs to know three things:

• How much of each resource could possibly request by each process.
• How much of each resource is currently holding by each process.
• How much of each resource the system currently has available.

Resources may be allocated to a process only if it satisfies the following conditions:

1. request ≤ max, else set error as process has crossed maximum claim made by it.
2. request ≤ available, else process waits until resources are available.

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] = 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]* = k, then process *Pi* may request at most k instances of resource type *Rj*.
- **Allocation:** *An n x m* matrix defines the number of resources of each type currently allocated to each process. If *Allocation[i,j]* = 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]* = k, then process Pi may need k more instances of resource type *Ri* to complete its task. Note that *Need[i,j] = Max[i,j] - Allocafion[i,j].*

These data structures vary over time in both size and value. The vector *Allocationi* specifies the resources currently allocated to process *Pi;* the vector *Needi* specifies the additional resources that process *Pi*
may still request to complete its task.

Safety Algorithm

The algorithm for finding out whether or not a system is in a safe state can be described as follows:
1. Let *Work* and *Finish* be vectors of length *m* and *n,* respectively. Initialize
Work := *Available* and *Finisk[i] :=false* for *i* = 1,2, ..., *n.*
2. Find an *i* such that both a. *Finisk[i] =false*
b. *Needi* < *Work.*
If no such ***i*** exists, go to step 4.
3. *Work* := *Work* + *Allocationi*
Finisk[i] := *true*
go to step 2.
4. If *Finish[i]* = *true* for all *i,* then the system is in a safe state.
This algorithm may require an order of *m* x *n2* operations to decide whether a state is safe.

Resource-Request Algorithm

Let *Requesti* be the request vector for process *Pi.* If *Requesti[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 *Requesti* < *Needi,* go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If *Requesti* < *Available,* 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 - Requesti;
Allocationi := Allocationi + Requesti;
Needi := Needi - Requesti;
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 *Requesti* and the old resource-allocation state is restored.

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
P0	0 1 0	7 5 3	3 3 2
PI	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

	Need
	A B C
P0	7 4 3
P1	1 2 2
P2	6 0 0
P3	0 1 1
P4	4 3 1

We claim that the system is currently in a safe state. Indeed, the sequence <PI, 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 Request1 = (1,0,2).

To decide whether this request can be immediately granted,we first check that Request1≤ Available (that is, (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
P0	0 1 0	7 4 3	2 3 0
PI	3 0 2	0 2 0	
P2	3 0 2	6 0 0	
P3	2 1 1	0 1 1	
P4	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 <PI, P3, P4, P0, P2> satisfies our safety requirement. Hence, we can immediately grant the request of process PI.

However, that when the system is in this state, a request for (3,3,0) by P4 cannot be granted, since the resources are not available. A request for (0,2,0) by Po cannot be granted, even though the resources are available, since the resulting state is unsafe.

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:

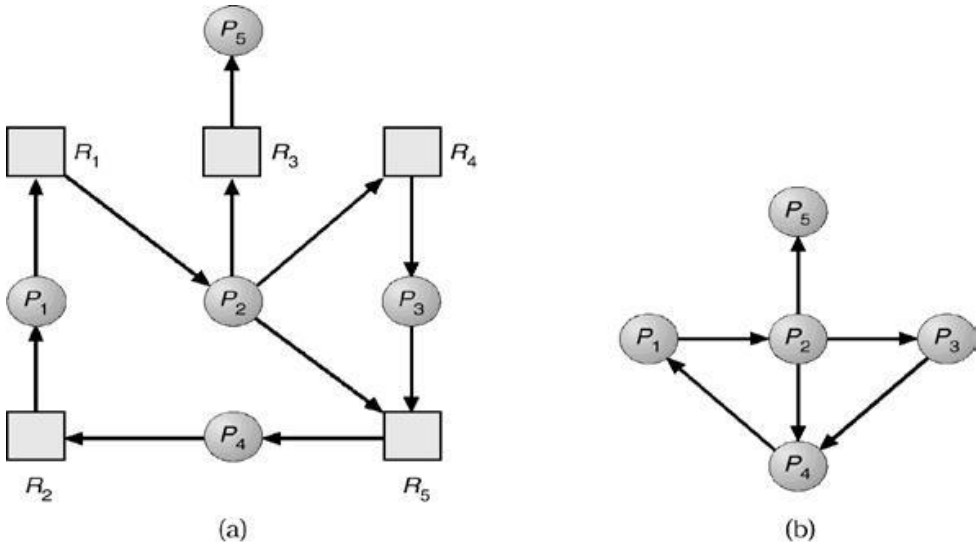
• An algorithm that examines the state of the system to determine whether a deadlock has occurred.
• An algorithm to recover from the deadlock.

According to number of instances in each resource type, the Deadlock Detection algorithm can be classified into two categories as follows:

1. Single Instance of Each Resource Type:

- If all resources have only a single instance, then it can define a deadlock detection algorithm that uses a variant of the resource-allocation graph (is called a *wait-for* graph).
- A wait–for graph can be draw by removing the nodes of type resource and collapsing the appropriate edges from the resource-allocation graph.
- 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.

For Example:



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 to 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.

2. Several Instances of a Resource Type: The following deadlock-detection algorithm is applicable to several instance of a resource type. The algorithm employs several time-varying data structures:

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 m matrix indicates the current request of each process. If Request[i,j] = k, then process *Pi* is requesting *k* more instances of resource type Rj.

The detection algorithm is described as follows:

1. Let *Work* and *Finish* be vectors of length *m* and *n,* respectively.

Initialize, Work := Available. For *i* = 1, 2, ..., *n,*

if *Allocationi* != 0, then *Finish[i] :=false;* otherwise, *Finish[i]* := *true.*

2. Find an index *i* such that both

a. *Finish[i] =false*

b. *Requesti < Work.*

If no such *i* exists, go to step 4.
3. *Work* := *Work* + *Allocationi*
Finish[i] := *true*
go to step 2.
4. If *Finish[i]* = false, for some *i,* 1 < *i* < *n,* then the system is in a deadlock state.
if *Finish[i] =false,* then process *Pi* is deadlocked.
This algorithm requires an order of *m* x *n2* operations to detect whether the system is in a deadlocked state.

RECOVERY FROM DEADLOCK
When a detection algorithm determines that a deadlock exists, then the system or operator is responsible for handling deadlock problem. There are two options for breaking a deadlock.
1. Process Termination
2. Resource preemption

Process Termination
There are two method to eliminate deadlocks by terminating a process as follows:
1. **Abort all deadlocked processes:** This method will break the deadlock cycle clearly by terminating all process. This method is cost effective. And it removes the partial computations completed by the processes.
2. **Abort one process at a time until the deadlock cycle is eliminated:** This method terminates one process at a time, and invokes a deadlock-detection algorithm to determine whether any processes are still deadlocked.
Resource Preemption: In resource preemption, the operator or system preempts 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:** The system or operator selects which resources and which processes are to be preempted based on cost factor.
2. **Rollback:** The system or operator must roll back the process to some safe state and restart it from that state.
3. **Starvation:** The system or operator should ensure that resources will not always be preempted from the same process?

FILE SYSTEM INTERFACE

- The 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:
  1.a collection of files, each storing related data, and
  2.a directory structure, which organizes and provides information about all the files in the system.

FILE CONCEPT

- A file is a collection of related information that is recorded on secondary storage.
- From a user's perspective, a file is the smallest allotment of logical secondary storage and data can not be written to secondary storage unless they are within a file.
- Four terms are in common use when discussing files: Field, Record, File and Database

- A **field** is the basic element of data. An individual field contains a single value, such as an employee's last name, a date, or the value of a sensor reading. It is characterized by its
- length and data type.
- A **record** is a collection of related fields that can be treated as a unit by some application program. For example, an employee record would contain such fields as name, social security number, job classification, date of hire, and so on.
- A **file** is a collection of similar records. The file is treated as a single entity by users and applications and may be referenced by name.
- A **database** is a collection of related data. A database may contain all of the information related to an organization or project, such as a business or a scientific study. The database itself consists of one or more types of files.

File Attributes:

A file has the following attributes:

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 those systems that support different types.

Location: This information is a pointer to a device and to the location of the file on that device.

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, modification and last use. These data can be useful for protection, security, and usage monitoring.

File Operations:

The operating system can provide system calls to create, write, read, reposition, delete, and truncate files. The file operations are described as followed:

Creating a file: Two steps are necessary to create a file.

1. First, space in the file system must be found for the file.
2. Second, an entry for the new file must be made in the directory. The directory entry records the name of the file and the location in the file system, and possibly other information

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 location of the file. 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.

Reading a file: To read from a file, we use a system call that specifies the name of the file and where (in main memory) the next block of the file should be put. Again, the directory is searched for the associated directory 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.

Repositioning within a file: The directory is searched for the appropriate entry, and the current-file-position is set to a given value. Repositioning within a file does not need to involve any actual I/O. This file operation is also known as a file *seeks.*

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.

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 file be reset to length zero and its file space released.

File Types: The files are classified into different categories as follows

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

The name is split into two parts-a name and an extension, The system uses the extension to indicate the type of the file and the type of operations that can be done on that file.

ACCESS METHODS

When a file is used, this information must be accessed and read into computer memory. The information in the file can be accessed in several ways. There are two major access methods as follows:

Sequential Access:

- Information in the file is processed in order, one record after the other.
- A read operation reads the next portion of the file and automatically advances a file pointer, which tracks the I/O location.
- Similarly, a write appends to the end of the file and advances to the end of the newly written material (the new end of file).
- Sequential access is based on a tape model of a file, and works as well on sequential-access devices as it does on random-access ones.

Direct 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.
- A direct-access file allows arbitrary blocks to be read or written.

- There are no restrictions on the order of reading or writing for a direct-access file.
- For the direct-access method, the file operations must be block number, rather than *read next,* and *write n* rather than *write next.*

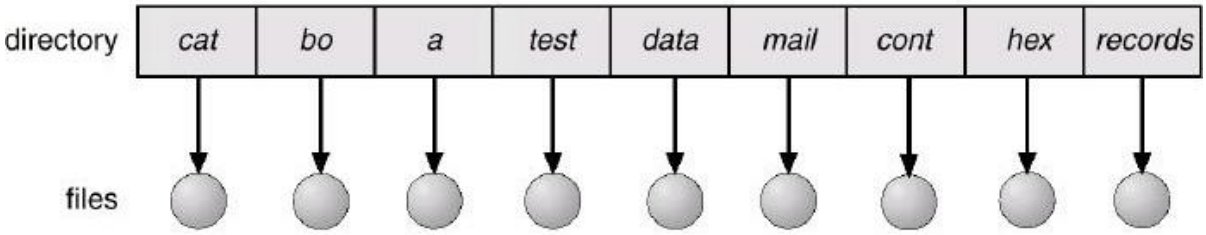
DIRECTORY STRUCTURE

A directory is an object that contains the names of file system objects. File system allows the users to organize files and other file system objects through the use of directories.The structure created by placement of names in directories can take a number of forms:

1. Single-level tree,
2. Two-level tree,
3. multi-level tree or cyclic graph.

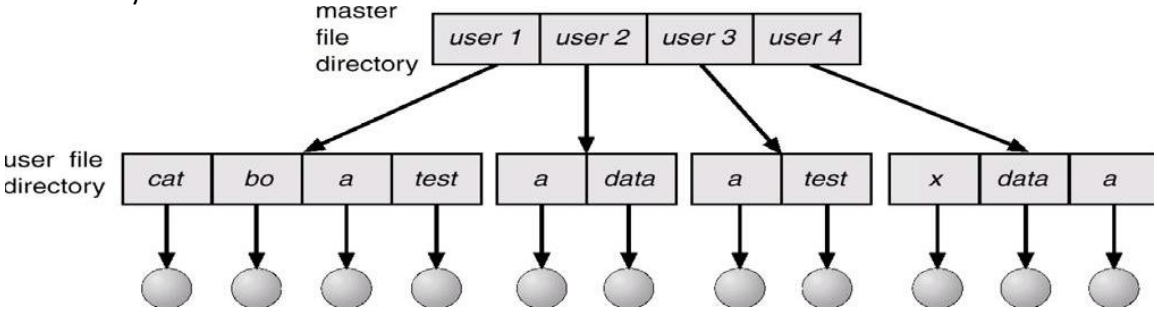
1. Single-Level Directory:

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



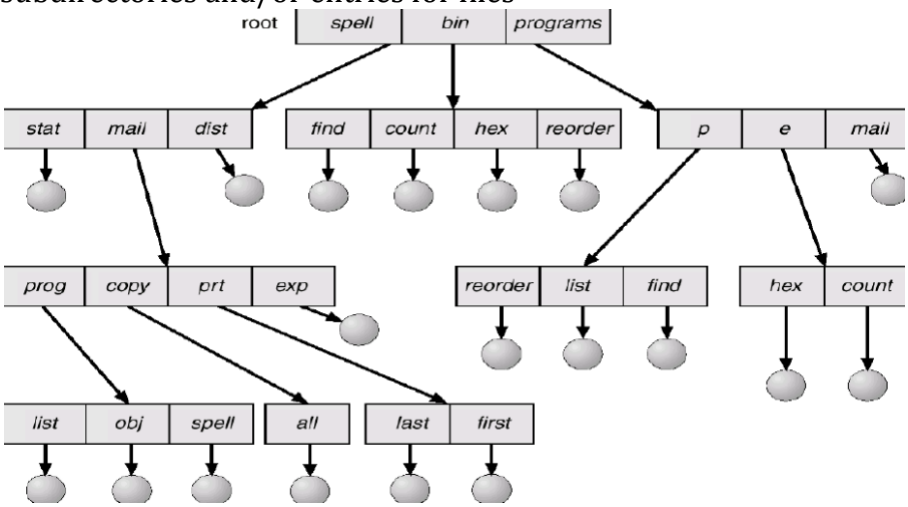
2. Two-Level Directory:

- In the two-level directory structure, each user has its own **user file directory (UFD).**
- Each UFD has a similar structure, but 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.
- 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.



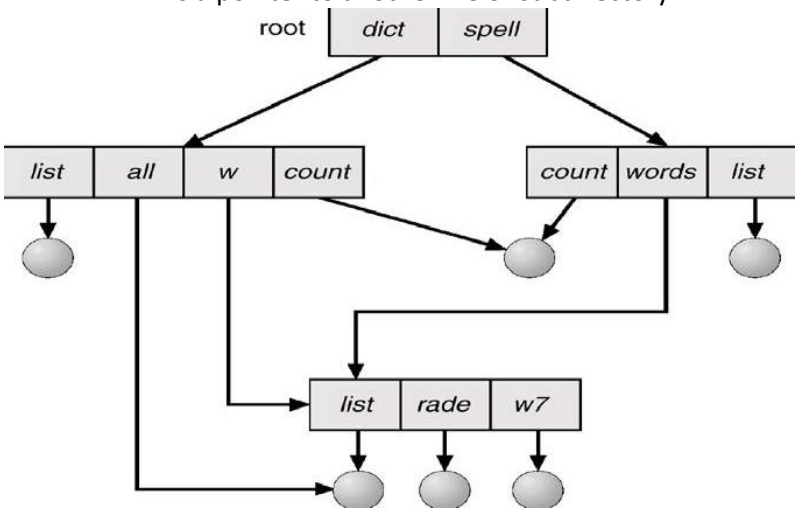
3. Tree-structured directories:

- A tree structure is A more powerful and flexible approach to organize files and directories in hierarchical.
- There is a master directory, which has under it a number of user directories.
- Each of these user directories may have subdirectories and files as entries.
- This is true at any level: That is, at any level, a directory may consist of entries for subdirectories and/or entries for files

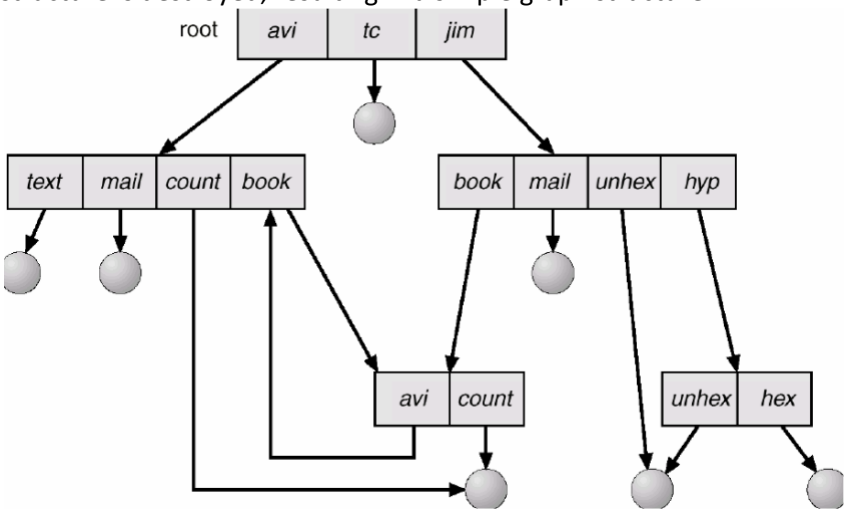


4. Acyclic-Graph Directories:

- An **acyclic graph** allows directories to have shared subdirectories and files.
- The *same* file or subdirectory may be in two different directories.
- An acyclic graph is a natural generalization of the tree structured directory scheme.
- A shared file (or directory) is not the same as two copies of the file.
- With two copies,each programmer can view the copy rather than the original, but if one programmer changes the file, the changes will not appear in the other's copy.
- Shared files and subdirectories can be implemented in several ways
- . A common way is to create a new directory entry called a link.
- A **link** is a pointer to another file or subdirectory



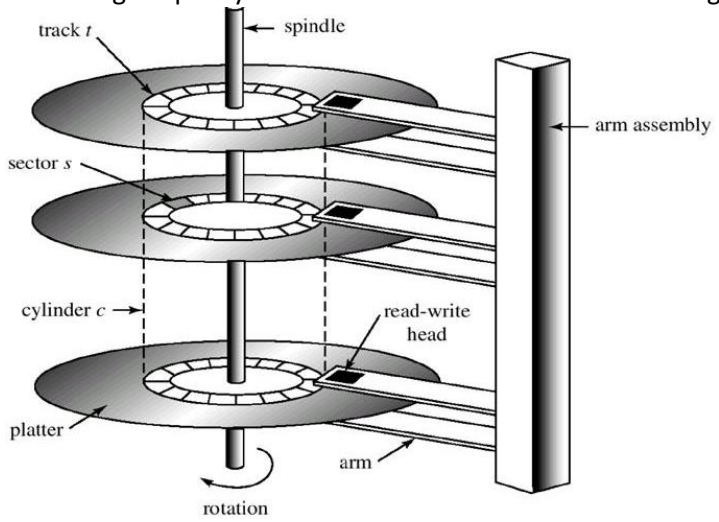
5. General Graph Directory: When we add links to an existing tree-structured directory, the tree structure is destroyed, resulting in a simple graph structure.



SECONDARY STORAGE STRUCTURE

DISKS STRUCTURE

- Magnetic disks provide the bulk of secondary storage for modern computer systems.
- Each disk **platter** has a flat circular shape, like a CD. Common platter diameters range from 1.8 to 5.25 inches.
- The two surfaces of a platter are covered with a magnetic material. We store information by recording it magnetically on the platters.
- A read-write head "flies" just above each surface of every platter.
- The heads are attached to a **disk arm,** which moves all the heads as a unit.
- The surface of a platter is logically divided into circular **tracks,** which are subdivided into **sectors.**
- The set of tracks that are at one arm position forms a **cylinder.**
- There may be thousands of concentric cylinders in a disk drive, and each track may contain hundreds of sectors.
- The storage capacity of common disk drives is measured in gigabytes.



- When the disk is in use, a drive motor spins it at high speed. Most drives rotate 60 to
- 200 times per second.

- Disk speed has two parts. The **transfer rate** is the rate at which data flow between the drive and the computer.
- The **positioning time (or random-access time)** consists of **seek time** and **rotational latency**.
- The **seek time is** the time to move the disk arm to the desired cylinder.
- And the **rotational latency is** the time for the desired sector to rotate to the disk head**.** Typical disks can transfer several megabytes of data per second and they have seek times and rotational latencies of several milliseconds.
- **Capacity of Magnetic disks(C) = S x T x P x N**
- Where S= no. of surfaces = 2 x no. of disks, T= no. of tracks in a surface,P= no. of sectors per track, N= size of each sector
- **Transfer Time:** The transfer time to or from the disk depends on the rotation speed of the disk in the following fashion:
- ***T* = b / (r x N)**
- Where *T*= transfer time, b=number of bytes to be transferred, *N*= number of bytes on a track, **r**= rotation speed, in revolutions per second.
- Modern disk drives are addressed as large one-dimensional arrays of **logical blocks,** where the logical block is the smallest unit of transfer.
- The one-dimensional array of logical blocks is mapped onto the sectors of the disk sequentially.
- Sector 0 is the first sector of the first track on the outermost cylinder.
- The mapping proceeds in order through that track, then through the rest of the tracks in that cylinder, and then through the rest of the cylinders from outermost to innermost.
- By using this mapping, we can convert a logical block number into an old-style disk address that consists of a cylinder number, a track number within that cylinder, and a sector number within that track.
- In practical, it is difficult to perform this translation, for two reasons.

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

- The density of bits per track is uniform. This is called **Constant linear velocity (CLV).**
- **T**he disk rotation speed can stay constant and the density of bits decreases from inner tracks to outer tracks to keep the data rate constant.
- This method is used in hard disks and is known as **constant angular velocity (CAV).**

DISK SCHEDULING

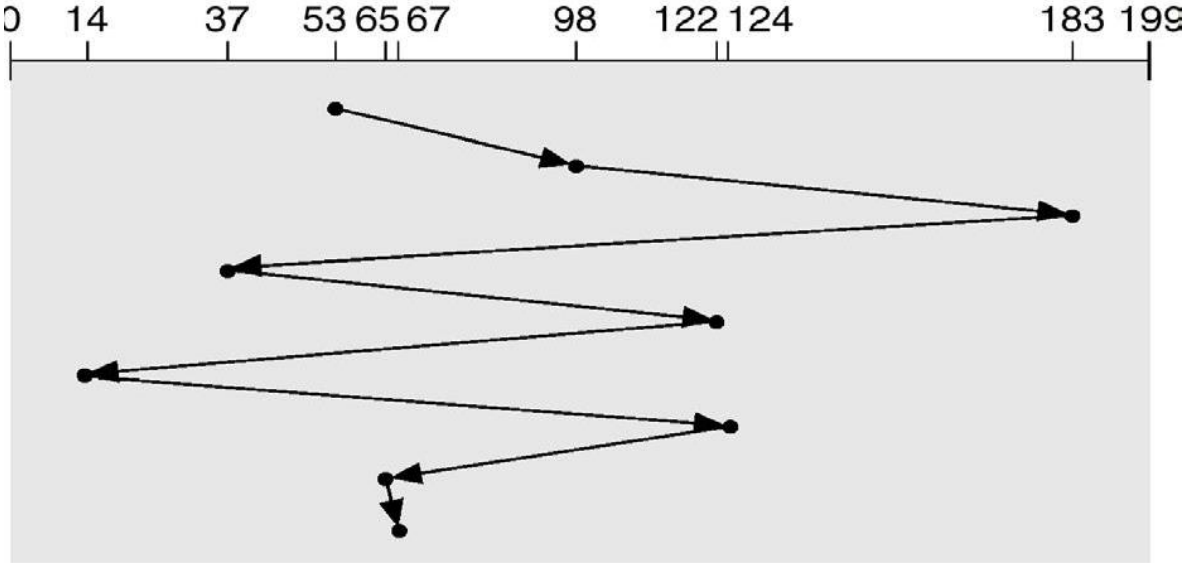
- The **seek time** is the time for the disk arm to move the heads to the cylinder containing the desired sector.
- The **rotational latency** is the time waiting for the disk to rotate the desired sector to the disk head.
- The disk **bandwidth** is the total number of bytes transferred divided by the total time between the first request for service and the completion of the last transfer. We can improve both the access time and the bandwidth by scheduling the servicing of disk I/O requests in a good order. Several algorithms exist to schedule the servicing of disk I/O requests as follows:

1. FCFS Scheduling

The simplest form of scheduling is first-in-first-out (FIFO) scheduling, which processes items from the queue in sequential order.

We illustrate this with a request queue (0-199): 98, 183, 37, 122, 14, 124, 65, 67

Consider now the Head pointer is in cylinder 53.



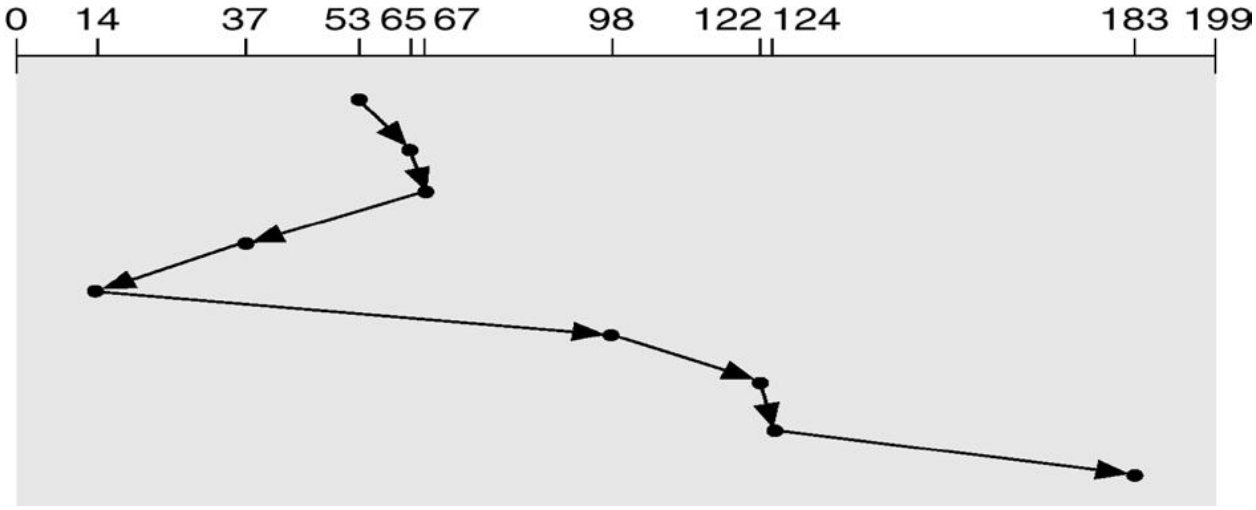
2. SSTF Scheduling

It stands for shortest-seek-time-first (SSTF) algorithm.

The SSTF algorithm selects the request with the minimum seek time from the current head position. Since seek time increases with the number of cylinders traversed by the head, SSTF chooses the pending request closest to the current head position. We illustrate this with a request queue (0-199):

98, 183, 37, 122, 14, 124, 65, 67

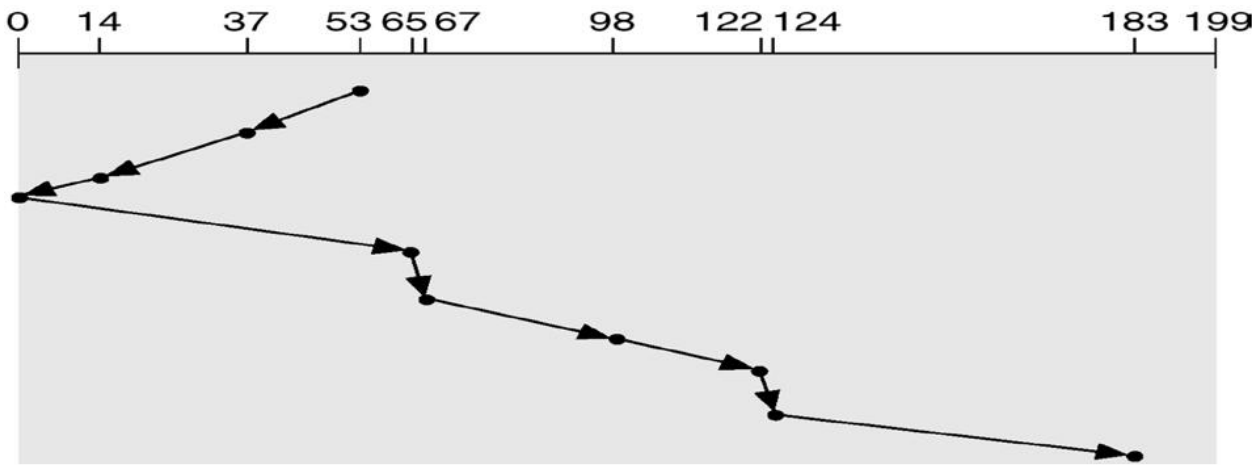
Consider now the Head pointer is in cylinder 53.



3. SCAN Scheduling

In the SCAN algorithm, the disk arm starts at one end of the disk, and moves toward the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk. At the other end, the direction of head movement is reversed, and servicing continues. The head continuously scans back and forth across the disk. We illustrate this with a request queue (0-199): 98, 183, 37, 122, 14, 124, 65, 67

Consider now the Head pointer is in cylinder 53.



4. C-SCAN Scheduling

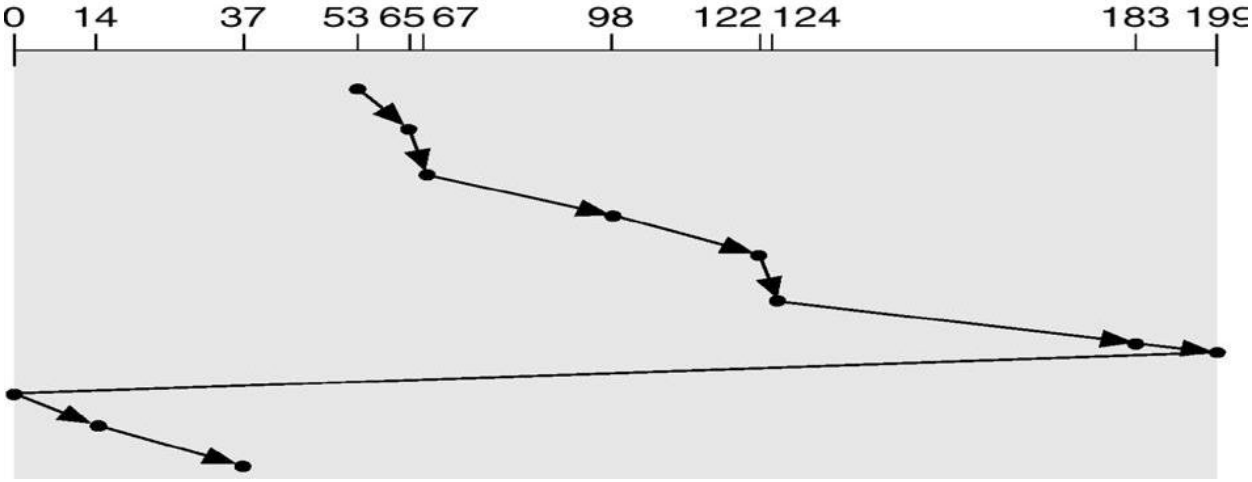
Circular **SCAN (C-SCAN)** scheduling is a variant of SCAN designed to provide a more uniform wait time. Like SCAN, C-SCAN moves the head from one end of the disk to the other, servicing requests along the way.

When the head reaches the other end, it immediately returns to the beginning of the disk, without servicing any requests on the return trip.

The CSCAN scheduling algorithm essentially treats the cylinders as a circular list that wraps around from the final cylinder to the first one. We illustrate this with a request queue (0-199):

98, 183, 37, 122, 14, 124, 65, 67.

Consider now the Head pointer is in cylinder 53.



5. LOOK Scheduling

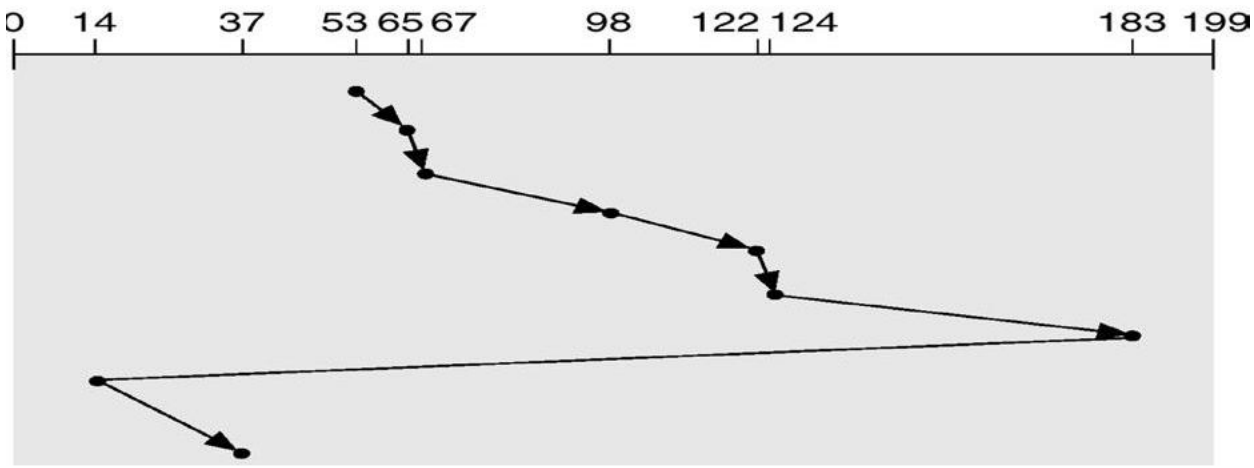
Practically, both SCAN and C-SCAN algorithm is not implemented this way.

More commonly, the arm goes only as far as the final request in each direction. Then, it reverses direction immediately, without going all the way to the end of the disk.

These versions of SCAN and C-SCAN are called **LOOK** and **C-LOOK** scheduling, because they look for a request before continuing to move in a given direction. We illustrate this with a request queue (0-199):

98, 183, 37, 122, 14, 124, 65, 67

Consider now the Head pointer is in cylinder 53.



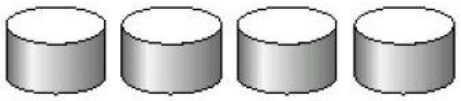
Selection of a Disk-Scheduling Algorithm

- SSTF is common and has a natural appeal.
- SCAN and C-SCAN perform better for systems that place a heavy load on the disk.
- Performance depends on the number and types of requests.
- Requests for disk service can be influenced by the file allocation method.
- The disk-scheduling algorithm should be written as a separate module of the operating
- system, allowing it to be replaced with a different algorithm if necessary.
- Either SSTF or LOOK is a reasonable choice for the default algorithm

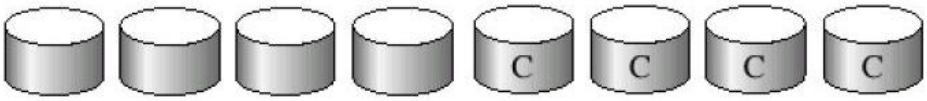
RAID STRUCTURE

A Redundant Array of Inexpensive Disks(RAID) may be used to increase disk reliability.
RAID may be implemented in hardware or in the operating system.
The RAID consists of seven levels, zero through six.
These levels designate different design architectures that share three common characteristics:

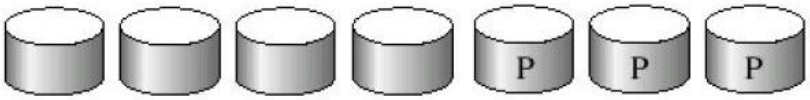
- RAID is a set of physical disk drives viewed by the operating system as a single logical drive.
- Data are distributed across the physical drives of an array in a scheme known as striping,
- Redundant disk capacity is used to store parity information, which guarantees data recoverability in case of a disk failure.



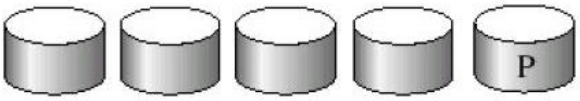
(a) RAID 0: non-redundant striping



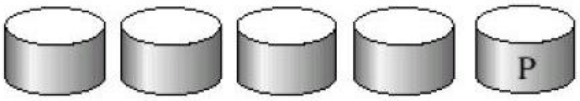
(b) RAID 1: mirrored disks



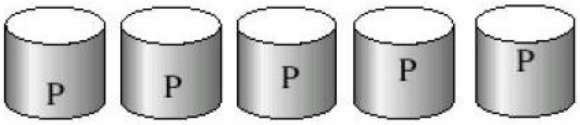
(c) RAID 2: memory-style error-correcting codes



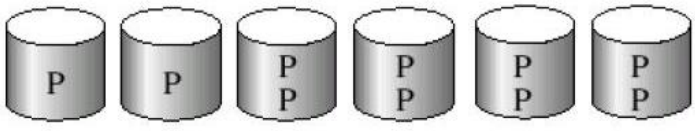
(d) RAID 3: bit-interleaved Parity



(e) RAID 4: block-interleaved parity



(f) RAID 5: block-lnterleaved distributed parity



(g) RAID 6: P + Q redundancy

(Here *P* indicates error-correcting bits and C indicates a second copy of the data)
The RAID levels are described as follows:
 RAID Level 0: RAID level 0 refers to disk arrays with striping at the level of blocks, but without any redundancy (such as parity bits). Figure(a) shows an array of size 4.

 RAID Level 1: RAID level 1 refers to disk mirroring. Figure (b) shows a mirrored organization that holds four disks' worth of data.
 RAID Level 2: RAID level 2 is also known as **memory-style error-correcting code (ECC) organization.** Each byte in a memory system may have a parity bit associated with it that records whether the numbers of bits in the byte set to 1 is even (parity=O) or odd (parity=l). The idea of ECC can be used directly in disk arrays via striping of bytes across
disks.
 RAID level 3: RAID level **3,** or **bit-interleaved parity organization,** improves on level 2 by noting that, disk controllers can detect whether a sector has been read correctly, so a single parity bit can be used for error correction, as well as for detection. The idea is as follows. If one of the sectors gets damaged, we know exactly which sector it is, and, for each bit in the sector, we can figure out whether it is a 1 or a 0 by computing the parity of the corresponding bits from sectors in the other disks. If the parity of the remaining bits is equal to the stored parity, the missing bit is 0; otherwise, it is 1.
 RAID Level 4: RAID level 4 or block-interleaved parity organization uses block-level striping, as in RAID 0 and in addition keeps a parity block on a separate disk for corresponding blocks from N other disks. This scheme is shown pictorially in Figure(e). If one of the disks fails, the parity block can be used with the corresponding blocks from the other disks to restore the blocks of the failed disk.
 RAID level 5: RAID level 5 or block-interleaved distributed parity is similar as level 4 but level 5 spreading data and parity among all N + 1 disks, rather than storing data in N disks and parity in one disk. For each block, one of the disks stores the parity, and the others store data. By spreading the parity across all the disks in the set, RAID 5 avoids the potential overuse of a single parity disk that can occur with RAID 4.
 RAID Level 6: RAID level 6 (is also called the P+Q redundancy scheme) is much like RAID level 5, but stores extra redundant information to guard against multiple disk failures. Instead of using parity, error-correcting codes such as the **Reed-Solomon codes** are used.

Stable Storage Implementation

By definition, information residing in the **Stable-Storage** is never lost. Even, if the disk and CPU have some errors, it will never lose any data.

Stable-Storage Implementation :

To achieve such storage, we need to replicate the required information on multiple storage devices with independent failure modes.
The writing of an update should be coordinate in such a way that it would not delete all the copies of the state and that, when we are recovering from a failure, we can force all the copies to a consistent and correct value, even if another failure occurs during the recovery
The disk write operation results to one of the following outcome:

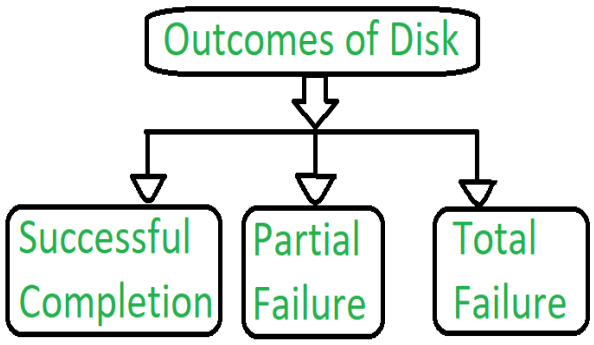


Figure – Outcomes of Disk

1. **Successful completion –**
   The data will written correctly on disk.
2. **Partial Failure –**
   In this case, failure is occurred in the middle of the data transfer, so that only some sectors were written with the new data, and the sector which is written during the failure may have been corrupted.
3. **Total Failure –**
   The failure occurred before the disk write started, so the previous data values on the disk remains intact.

During writing a block somehow if failure occurs, the system's first work is to detect the failure and then invoke a recovery process to restore the consistent state. To do that, the system must contain two physical block for each logical block.

An output operation is executed as follows:

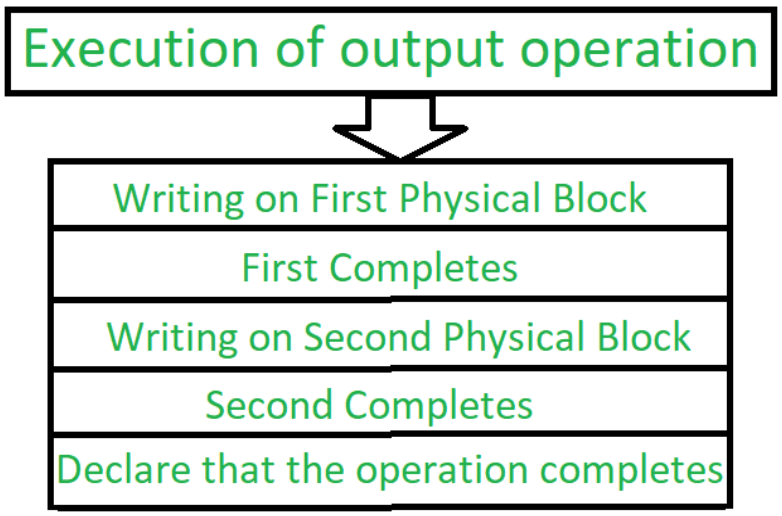


Figure – Process of execution of output operation

1. Write the information onto the first physical block.
2. When the first write completes successfully, write the same operation onto the second physical block.
3. When both the operation declares successfully, declare the operation as complete.

During the recovery from a failure each of the physical block is examined. If both are the same and no detectable error exists, then no further action is necessary. If one block contains detectable errors then we replace its content with the value of the other block. If neither block contains the detectable error, but the block differ in content, then we replace the content of first block with the content of

the second block.This procedure of the recovery give us an conclusion that either the write to stable content succeeds successfully or it results in no change.

This procedure will be extended if we want arbitrarily large number of copies of each block of the stable storage. With the usage of large number of copies, the chances of the failure reduces. Generally, it is usually reasonable to simulate stable storage with only two copies. The data present in the stable storage is safe unless a failure destroys all the copies. The data that is present in the stable storage is guaranteed to be safe unless a failure destroys all the copies.

Because waiting for disk writes to complete is time consuming, many storage arrays add NVRAM as a cache. Since the memory is no-volatile it can be trusted to store the data en route to the disks. In this way it is considered as a part of the stable storage. Writing to the stable storage is much faster than to disk, so performance is greatly improved.