UNIT 4

<u>Deadlocks</u>: Resources, Conditions for resource deadlocks, Ostrich algorithm, Deadlock detection And recovery, Deadlock avoidance, Deadlock prevention.

<u>File Systems</u>: Files, Directories, File system implementation, management and optimization. Secondary-Storage Structure: Overviewof disk structure, and attachment, Disk scheduling, RAID structure, Stable storage implementation.

DEADLOCKS

<u>Definition</u>

In a multiprogramming system, processes request resources. If those resources are being used by other processes then the process enters a waiting state. However, if other processes are also in a waiting state, we have deadlock.

The formal definition of deadlock is as follows:

Definition: A set of processes is in a deadlock state if every process in the set is waiting for an event (release) that can only be caused by some other process in the same set.

Example

Process-1 requests the printer, gets it
Process-2 requests the tape unit, gets it Process-1 and
Process-1 requests the tape unit, waits Process-2 are
Process-2 requests the printer, waits deadlocked!

we shall analyze deadlocks with the following assumptions:

• A process must request a resource before using it. It must release the resource after using it.

- request
- use
- release

• A process cannot request a number more than the total number of resources available in the system.

<u>Deadlock Characterization</u>

A deadlock occurs if and only if the following four conditions hold in a system simultaneously:

1. Mutual Exclusion: At least one of the resources is non-sharable (that is; only a limited number of processes can use it at a time and if it is requested by a process while it is being used by another one, the requesting process has to wait until the resource is released.).

2. Hold and Wait: There must be at least one process that is holding at least one resource and waiting for other resources that are being hold by other processes.

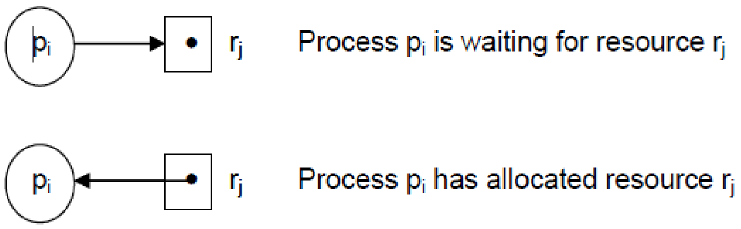
3. No Preemption: No resource can be preempted before the holding process completes its task with that resource.

4. Circular Wait: There exists a set of processes: {P1, P2, ..., Pn} such that

P1 is waiting for a resource held by P2
P2 is waiting for a resource held by P3
...
Pn-1 is waiting for a resource held by Pn
Pn is waiting for a resource held by P1

Resource Allocation Graphs

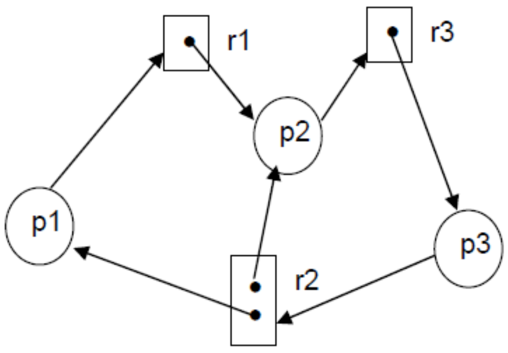
Resource allocation graphs are drawn in order to see the allocation relations of processes and resources easily. In these graphs, processes are represented by circles and resources are represented by boxes. Resource boxes have some number of dots inside indicating available number of that resource, that is number of instances.



p_i
r_j
Process p_i is waiting for resource r_j
p_i
r_j
Process p_i has allocated resource r_j

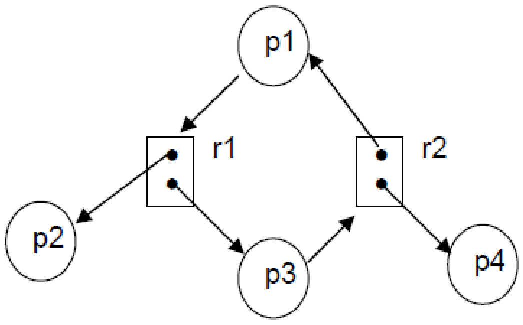
• If the resource allocation graph contains no cycles then there is no deadlock in the system at that instance.
• If the resource allocation graph contains a cycle then a deadlock may exist.
• If there is a cycle, and the cycle involves only resources which have a single instance, then a deadlock has occurred.

Example



There are three cycles, so a deadlock may exists. Actually p1, p2 and p3 are deadlocked

Example



There is a cycle, however there is no deadlock. If p4 releases r2, r2 may be allocated to p3, which breaks the cycle.

Deadlock Prevention

To prevent the system from deadlocks, one of the four discussed conditions that may create a deadlock should be discarded. The methods for those conditions are as follows:

Mutual Exclusion:

In general, we do not have systems with all resources being sharable. Some resources like printers, processing units are non-sharable. So it is not possible to prevent deadlocks by denying mutual exclusion.

Hold and Wait:

One protocol to ensure that hold-and-wait condition never occurs says each process must request and get all of its resources before it begins execution.
Another protocol is “Each process can request resources only when it does not occupies any resources.”
The second protocol is better. However, both protocols cause low resource utilization and starvation. Many resources are allocated but most of them are unused for a long period of time. A process that requests several commonly used resources causes many others to wait indefinitely.

No Preemption:

One protocol is “If a process that is holding some resources requests another resource and that resource cannot be allocated to it, then it must release all resources that are currently allocated to it.”
Another protocol is “When a process requests some resources, if they are available, allocate them. If a resource it requested is not available, then we check whether it is being used or it is allocated to some other process waiting for other resources. If that resource is not being used, then the OS preempts it from the waiting process and allocate it to the requesting process. If that resource is used, the requesting process must wait.” This protocol can be applied to resources whose states can easily be saved and restored (registers, memory space). It cannot be applied to resources like printers.

Circular Wait:

One protocol to ensure that the circular wait condition never holds is “Impose a linear ordering of all resource types.” Then, each process can only request resources in an increasing order of priority.
For example,
set priorities for r1 = 1, r2 = 2, r3 = 3, and r4 = 4.
With these priorities, if process P wants to use r1 and r3, it should first request r1, then r3. Another protocol is “Whenever a process requests a resource rj, it must have released all resources rk with priority(rk) ≥ priority (rj).

Deadlock avoidance

Given some additional information on how each process will request resources, it is possible to construct an algorithm that will avoid deadlock states. The algorithm will dynamically examine the resource allocation operations to ensure that there won't be a circular wait on resources.
When a process requests a resource that is already available, the system must decide whether that resource can immediately be allocated or not. The resource is immediately allocated only if it leaves the system in a *safe state*.
A state is safe if the system can allocate resources to each process in some order avoiding a deadlock. A deadlock state is an unsafe state.

Example

Consider a system with 12 tape drives. Assume there are three processes : p1, p2, p3. Assume we know the maximum number of tape drives that each process may request:

p1 : 10, p2 : 4, p3 : 9

Suppose at time t_{now}, 9 tape drives are allocated as follows :

p1 : 5, p2 : 2, p3 : 2

So, we have three more tape drives which are free.
This system is in a safe state because it we sequence processes as: <p2, p1, p3>, then p2 can get two more tape drives and it finishes its job, and returns four tape drives to the system. Then the system will have 5 free tape drives. Allocate all of them to p1, it gets 10 tape drives and finishes its job. p1 then returns all 10 drives to the system. Then p3 can get 7 more tape drives and it does its job.
It is possible to go from a safe state to an unsafe state:

Example

Consider the above example. At time tnow+1, p3 requests one more tape drive and gets it. Now, the system is in an unsafe state.
There are two free tape drives, so only p2 can be allocated all its tape drives. When it finishes and returns all 4 tape drives, the system will have four free tape drives.
p1 is allocated 5, may request 5 more → has to wait
p3 is allocated 3, may request 6 more → has to wait
We allocated p3 one more tape drive and this caused a deadlock.

Banker's Algorithm (Dijkstra and Habermann)

It is a deadlock avoidance algorithm. The following data structures are used in the algorithm:

m = number of resources
n = number of processes

Available [m] : One dimensional array of size m. It indicates the number of available resources of each type. For example, if Available [i] is k, there are k instances of resource ri.

Max [n,m] : Two dimensional array of size n*m. It defines the maximum demand of each process from each resource type. For example, if Max [i,j] is k, process pi may request at most k instances of resource type rj.

Allocation [n,m] : Two dimensional array of size n*m. It defines the number of resources of each type currently allocated to each process.

Need [n,m] : Two dimensional array of size n*m. It indicates the remaining need of each process, of each resource type. If Need [i,j] is k, process pi may need k more instances of resource type rj. Note that Need [i,j] = Max [i,j] - Allocation [i,j].

Request [n,m] : Two dimensional array of size n*m. It indicates the pending requests of each process, of each resource type.
Now, take each row vector in Allocation and Need as Allocation(i) and Need(i). (Allocation(i)
specifies the resources currently allocated to process pi.)
Define the ≤ relation between two vectors X and Y , of equal size = n as :

$$X \leq Y \Leftrightarrow X[i] \leq Y[i],\ i = 1,2, ..., n$$

$X \not\leq Y \Leftrightarrow X[i] > Y[i]$ for some i

The algorithm is as follows:

1. Process pi makes requests for resources. Let Request(i) be the corresponding request vector. So, if pi wants k instances of resource type rj, then Request(i)[j] = k.

2. If Request(i) !≤ Need(i), there is an error.

3. Otherwise, if Request(i) !≤ Available, then pi must wait.

4. Otherwise, Modify the data structures as follows :

```
Available = Available - Request(i)
Allocation(i) = Allocation(i) + Request(i)
Need(i) = Need(i) - Request(i)
```

5. Check whether the resulting state is safe. (Use the safety algorithm presented below.)

6. If the state is safe, do the allocation. Otherwise, pi must wait for Request(i).

Safety Algorithm to perform Step 5:

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

```
1. Initialize Work = Available, Finish [j] = false, for all j.
2. Find an i such that Finish [ i ] = false and Need(i) ≤ Work
     If no such i is found, go to step 4.
3. If an i is found, then for that i, do :
         Work = Work + Allocation(i)
         Finish [i] = true
     Go to step 2.
4. If Finish [j] = true for all j, then the system is in a safe state.
```

Banker's algorithm is O(m × (n2)).

Example 5.6*:* (Banker's algorithm)

Given

$$\text{Available} = \begin{bmatrix} 1 & 4 & 1 \end{bmatrix}$$

$$\text{Max} = \begin{bmatrix} 1 & 3 & 1 \\ 1 & 4 & 1 \end{bmatrix}$$

$$\text{Allocation} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

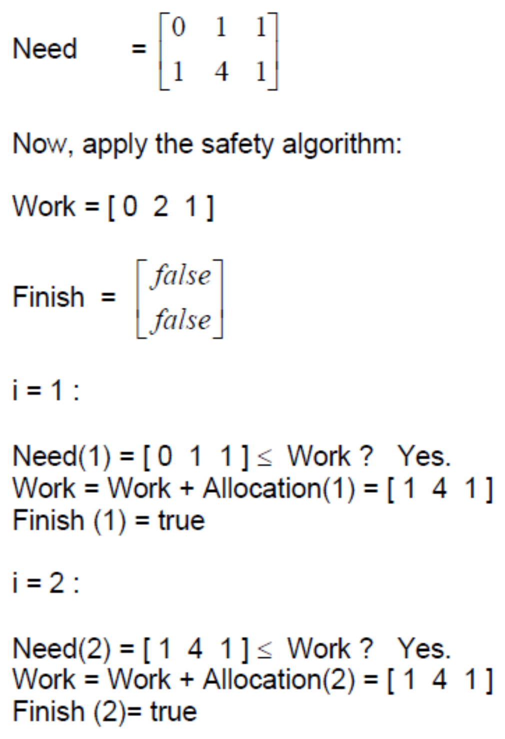
$$\text{Need} = \begin{bmatrix} 1 & 3 & 1 \\ 1 & 4 & 1 \end{bmatrix}$$

$$\text{Request} = \begin{bmatrix} 1 & 2 & 0 \\ 0 & 2 & 1 \end{bmatrix}$$

Request(1) is to be processed. If it is satisfied data would become:

$$\text{Available} = \begin{bmatrix} 0 & 2 & 1 \end{bmatrix}$$

$$\text{Allocation} = \begin{bmatrix} 1 & 2 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$



Need $= \begin{bmatrix} 0 & 1 & 1 \\ 1 & 4 & 1 \end{bmatrix}$

Now, apply the safety algorithm:

Work = [0 2 1]

Finish $= \begin{bmatrix} false \\ false \end{bmatrix}$

i = 1 :

Need(1) = [0 1 1] $\leq$ Work ? Yes.
Work = Work + Allocation(1) = [1 4 1]
Finish (1) = true

i = 2 :

Need(2) = [1 4 1] $\leq$ Work ? Yes.
Work = Work + Allocation(2) = [1 4 1]
Finish (2)= true

System is in a safe state, so do the allocation. If the algorithm is repeated for Request(2), the system will end up in an unsafe state.

Deadlock Detection

If a system has no deadlock prevention and no deadlock avoidance scheme, then it needs a deadlock detection scheme with recovery from deadlock capability. For this, information should be kept on the allocation of resources to processes, and on outstanding allocation requests. Then, an algorithm is needed which will determine whether the system has entered a deadlock state. This algorithm must be invoked periodically.

Deadlock Detection Algorithm (Shoshani and Coffman)

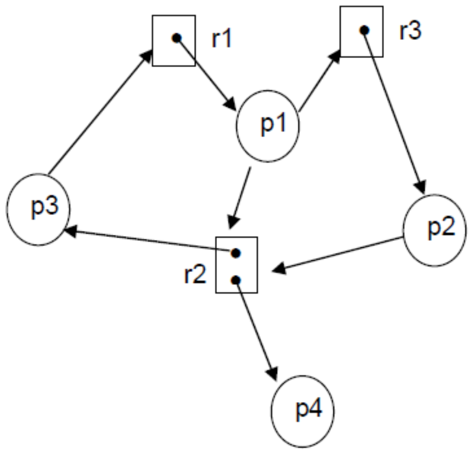
Data Structure is as:
Available [m]
Allocation [n,m] as in Banker's Algorithm
Request [n,m] indicates the current requests of each process.
Let Work and Finish be vectors of length m and n, as in the safety algorithm.
The algorithm is as follows:

```
1. Initialize Work = Available
        For i = 1 to n do
                If Allocation(i) = 0 then
                        Finish[i] = true
                else
                        Finish[i] = false
2. Search an i such that
                Finish[i] = false and Request(i) ≤ Work
        If no such i can be found, go to step 4.
```

```
3. For that i found in step 2 do:
        Work = Work + Allocation(i)
        Finish[i] = true
    Go to step 2.
4. If Finish[i] ≠ true for a some i then
    the system is in deadlock state
  else
    the system is safe
```

Example

Examine whether the system whose resource allocation graph is given below is deadlocked or not.



First, let' s form the required structures:

Available = [0 0 0]

$$\text{Allocation} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\text{Request} = \begin{bmatrix} 0 & 1 & 1 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

$$\text{Finish} = \begin{bmatrix} False \\ False \\ False \\ False \end{bmatrix}$$

Work = [0 0 0]

Request(4) ≤ Work → i = 4:

Work = Work + Allocation(4) = [0 0 0] + [0 0 1] = [0 0 1] ;

Finish[4] = True

Request(2) ≤ Work → i = 2:

Work = Work + Allocation(2) = [0 0 1] + [0 1 0] = [0 1 1] ;

Finish[2] = True

Request(1) ≤ Work→ i = 1:

Work = Work + Allocation(1) = [0 1 1] + [1 0 0] = [1 1 1] ;
Finish[1] = True
Request(3) $\leq$ Work $\rightarrow$ i = 3:
Work = Work + Allocation(3) = [1 1 1] + [0 0 1] = [1 1 2] ;
Finish[3] = True
Since Finish[i] = true for all i, there is no deadlock in the system

Recovery From Deadlock
If the system is in a deadlock state, some methods for recovering it from the deadlock state must be applied. There are various ways for recovery:

- Allocate one resource to several processes, by violating mutual exclusion.
- Preempt some resources from some of the deadlocked processes.
- Abort one or more processes in order to break the deadlock.

If preemption is used:

1. Select a victim. (Which resource(s) is/are to be preempted from which process?)
2. Rollback: If we preempt a resource from a process, roll the process back to some safe state and mak it continue.

Here the OS may be probably encounter the problem of starvation. How can we guarantee that resources will not always be preempted from the same process?
In selecting a victim, important parameters are:

- Process priorities
- How long the process has occupied?
- How long will it occupy to finish its job
- How many resources of what type did the process use?
- How many more resources does the process need to finish its job?
- How many processes will be rolled back? (More than one victim may be selected.)

For rollback, the simplest solution is a total rollback. A better solution is to roll the victim process back only as far as it’s necessary to break the deadlock. However, the OS needs to keep more information about process states to use the second solution.

To avoid starvation, ensure that a process can be picked as a victim for only a small number of times. So, it is a wise idea to include the number of rollbacks as a parameter.

<u>File Systems</u>: Files, Directories, File system implementation, management and optimization. Secondary-Storage Structure: Overviewof disk structure, and attachment, Disk scheduling, RAID structure, Stable storage implementation

FILE CONCEPT

A file is a named collection of related information that is recorded on secondary storage.

 A file is the smallest allotment of logical secondary storage (i.e.) data cannot be written to secondary storage unless they are within a file.

 Files represent programs and data. Data files may be numeric, alphanumeric or binary.

 The information in a file is defined by its creator.

 Different types of information may be stored in a file such as source or executable programs, numeric or text data, photos, music, video and so on.

A file structure depends on its type:

 Text file is a sequence of characters organized into lines.

 Source file is a sequence of functions, each of which is further organized as declarations followed by executable statements.

 Executable file is a series of code sections that the loader can bring into memory and execute.

File Attributes

A file is referred to by its name. The following are the list of file attributes:

 Name: The symbolic file name is the only information kept in human-readable form.

 Identifier: This is a unique number that 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: It is a pointer to a device and to the location of the file on that device.

 Size: The current size of the file and the maximum size are included in this attribute.

 Protection: It is access-control information determines who can do reading, writing, executing and so on.

 Time, Date and User Identification: This information kept for creation, last modification and last use. These data can be useful for protection, security and usage monitoring.

Directory structure keeps information about all files. It resides on secondary storage.

 A directory entry consists of the file's name and its unique identifier.

 The identifier locates the other file attributes.

 It may take more than a kilobyte to record this information for each file.

File Operations

There are 6 basic operations performed on file and corresponding System call are:

1. **Creating a file: create( )** system call is used to create a file. To create a file Operating system checks whether there is enough space in the system. If yes, then a new entry will be made in the directory structure.

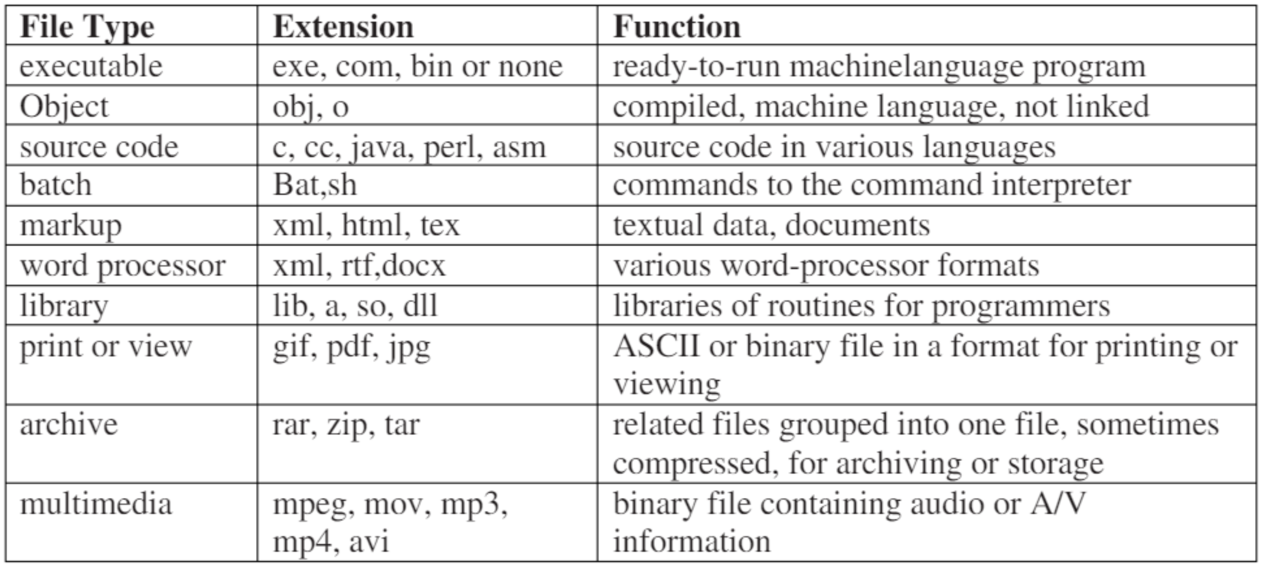
2. Repositioning within a file. The directory is searched for the appropriate entry and the current-file-position pointer is repositioned to a given value. Repositioning within a file need not involve any actual I/O. This file operation is also known as a File-Seek.

3. Deleting a file: delete( ) system call is used to delete a file. To deete a file, we search the directory for the named file. If we found the associated directory entry, we release all file space and erase the directory entry.

4. Truncating a file. The user erases all the contents of a file but keep its attributes. The length of the file will be reset to zero.
5. Writing a file. write( ) system call is used to write a file. It specifies both the name of the file and the information to be written to the file. The system searches the filename in the directory to find the file's location.
6. Reading a file. read( ) system call is used to read from a file. It specifies the name of the file and where the next block of the file should be put. The directory is searched for the associated entry.

File Types

□ File types are generally included as part of file name. The file name is split into two parts: a name and an extension usually separated by a dot.
□ The system uses the extension to indicate the type of the file and the type of operations that can be done on that file.
Example: resume.docx, server.c and ReaderThread.cpp.
The below table shows the common file types:



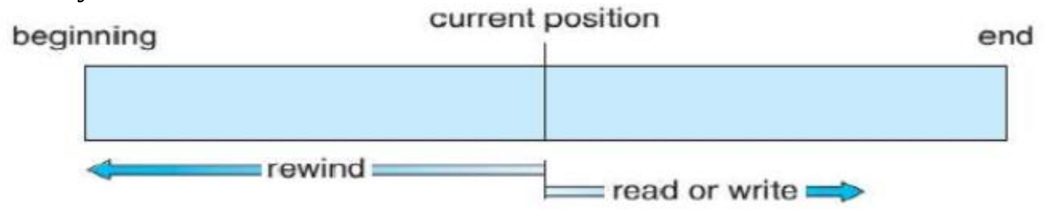
File Type	Extension	Function
executable	exe, com, bin or none	ready-to-run machinelanguage program
Object	obj, o	compiled, machine language, not linked
source code	c, cc, java, perl, asm	source code in various languages
batch	Bat,sh	commands to the command interpreter
markup	xml, html, tex	textual data, documents
word processor	xml, rtf,docx	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	gif, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	rar, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, mp3, mp4, avi	binary file containing audio or A/V information

information Internal File Structure
□ Locating an offset within a file can be complicated for the operating system.
□ Disk systems 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). All blocks are the same size.

ACCESS METHODS

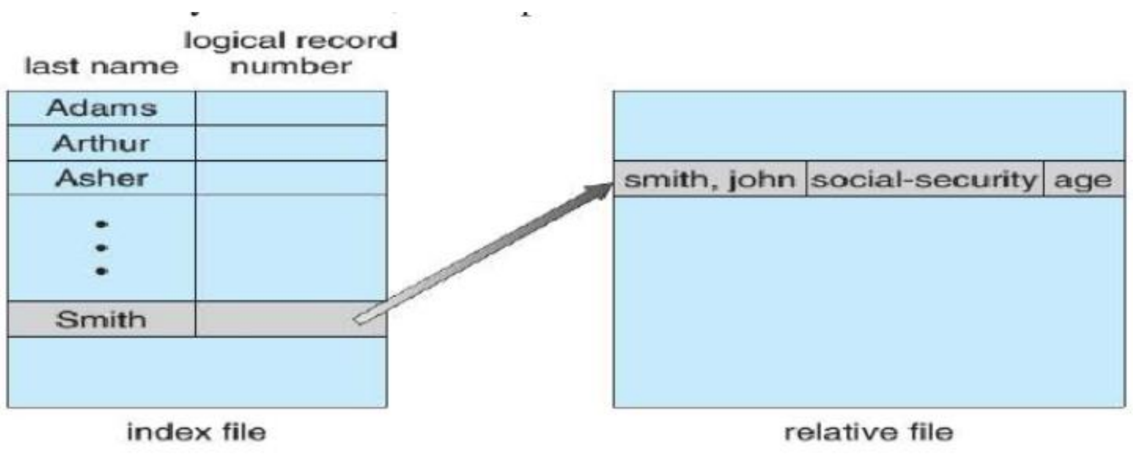
Files store information. When a file is used, this information must be accessed and read into computer memory. The information in the file can be accessed in several ways:
1. Sequential Access
2. Direct Access
3. Indexed Access
Sequential Access Information in the file is processed in order, one record after the other record. Example: editors and compilers usually access files in sequential order. Reads and writes make up the bulk of operations on a file:

 read_next( ) operation reads the next portion of the file and automatically advances a file pointer, which tracks the I/O location.
 write_next( ) operation appends to the end of the file and advances to the end of the newly written material.



Sequential access is based on a tape model of a file and works on sequential-access devices.

<u>Direct Access or Relative Access</u>
 In direct access method, the file is viewed as a numbered sequence of blocks or records.
 There are no restrictions on the order of reading or writing for a direct-access file.
 Thus, we may read block 14, then read block 53 and then write block 7.
 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.
 Databases are direct access type. When a query concerning a particular subject arrives, we compute which block contains the answer and then read that block directly to provide the desired information.
Example: Airline-reservation system
 We might store all the information about a particular flight 713 in the block identified by the flight number.
 The number of available seats for flight 713 is stored in block 713 of the reservation file.
 To store information about a larger set, such as people, we might compute a hash function on the people’s names to determine a block to read and search.
File operation in Direct Access Method
read(n) and write(n) are the read and write operation performed in Direct Access method where n represent the block number.
The block number provided by the user to the operating system is 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.
 Relative block numbers allows the OS to decide where the file should be placed and helps to prevent user from accessing portions of the file system that may not be part of its file.
Indexed Access
The index is like an index in the back of a book that 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.
 To find a record we can make a binary search of the index. This search helps us to know exactly which block contains the desired record and access that block.
 This structure allows us to search a large file doing little I/O.
 With large files, the index file itself may become too large to be kept in memory.
 One solution is to create an index for the index file. The primary index file contains pointers to secondary index files, which point to the actual data items.



DIRECTORY AND DISK STRUCTURE

Files are stored on random-access storage devices such as Hard-disks, Optical-disks and Solid-state disks.

 A storage device can be used for a file system. It can be subdivided for finer-grained control.

 Ex: A disk can be partitioned into quarters. Each quarter can hold a separate file system.

 Partitioning is useful for limiting the sizes of individual file systems, putting multiple file-system types on the same device or leaving part of the device available for other uses, such as swap space or unformatted (raw) disk space.

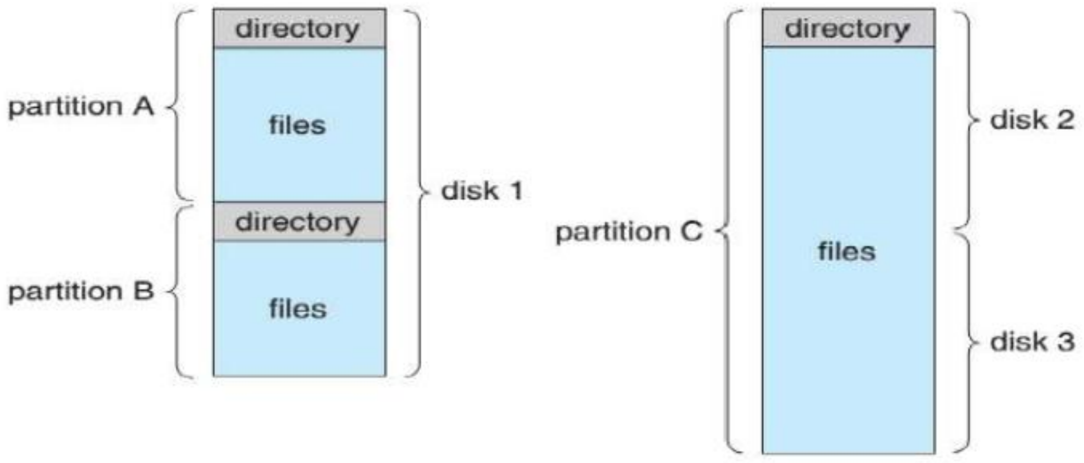
 A file system can be created on each of these disk partitions. Any entity containing a file system is generally known as a Volume.

 Volume may be a subset of a device, a whole device. Each volume can be thought of as a virtual disk. Volumes can also store multiple operating systems. Volumes allow a system to boot and run more than one operating system.

 Each volume contains a file system maintains 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 (directory) records information such as name, location, size and type for all files on that volume. The below figure shows the typical file system organization:



Storage Structure in Solaris OS

The file systems of computers can be extensive. Even within a file system, it is useful to segregate files into groups and manage those groups. This organization involves the use of directories.

Consider the types of file systems in the Solaris Operating system:

- Tmpfs is a ―temporary‖ file system that is created in volatile main memory and has its contents erased if the system reboots or crashes
- objfs is a ―virtual‖ file system that gives debuggers access to kernel symbols
- ctfs is a virtual file system that maintains ―contract‖ information to manage which processes start when the system boots and must continue to run during operation
- lofs is a ―loop back‖ file system that allows one file system to be accessed in place of another file system.
- procfs is a virtual file system that presents information on all processes as a file system
- ufs, zfs are general-purpose file systems.

Operations on Directory

Different operations performed on a directory are:

- Search for a file. This operation searches a directory structure to find the entry for a particular file. It finds all files whose names match with a particular pattern.
- Create a file. When a new file is created its entry is added to the directory.
- Delete a file. When a file is no longer needed, we can remove it from the directory.
- 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.
- Rename a file. A file can be renamed, when the contents or use of the file changes (i.e.) csec.txt to cse.txt or cse.txt to cse.c file etc.
- Traverse the file system. We may wish to access every directory and every file within a directory structure.

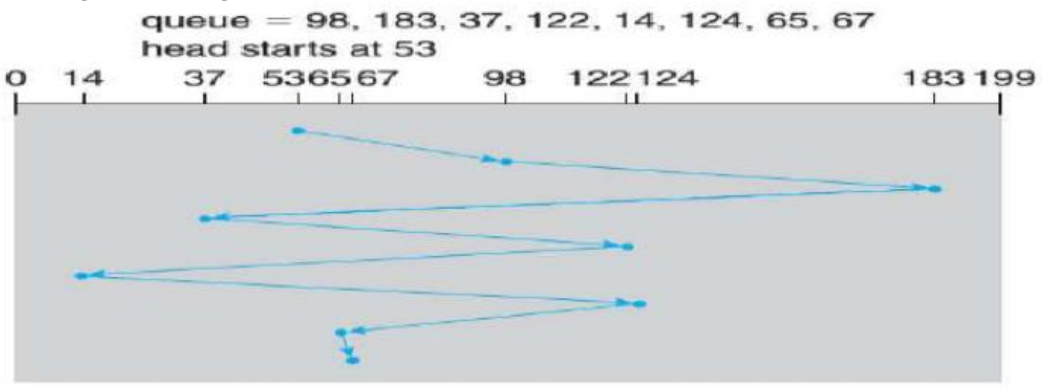
DISK SCHEDULING ALGORITHMS

One of the responsibilities of the operating system is to use the hardware efficiently. 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.

Different Disk Scheduling algorithms are:

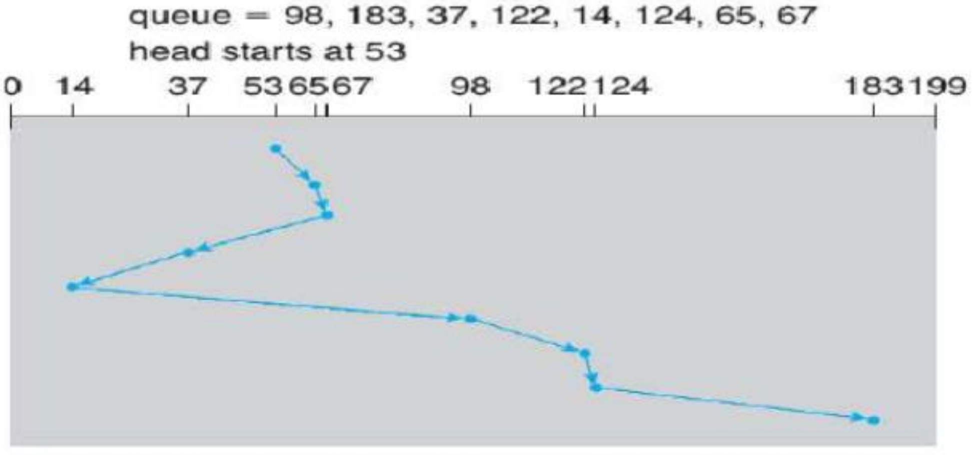
1. FCFS Scheduling
2. SSTF Scheduling
3. SCAN Scheduling
4. C- SCAN Scheduling
5. LOOK Scheduling

First-Come-First-Serve Algorithm
FCFS does not provide the fastest service. Consider a disk queue with requests for I/O to blocks on cylinders in the order: 98, 183, 37, 122, 14, 124, 65, 67 The disk head is initially at cylinder 53. By using FCFS algorithm:



 It will first move from 53 to 98 the head movement of 45 cylinders
 Then 98 to 183 the head movement of 85 cylinders
 Then 183 to 37 the head movement of 146 cylinders
 Then 37 to 122 the head movement of 85 cylinders
 Then 122 to 14 the head movement of 108 cylinders
 Then 14 to 124 the head movement of 110 cylinders
 Then 124 to 65 the head movement of 59 cylinders
 Then 65 to 67 the head movement of 2 cylinders
 The total head movement of 640 cylinders.
FCFS algorithm reduces the system performance.

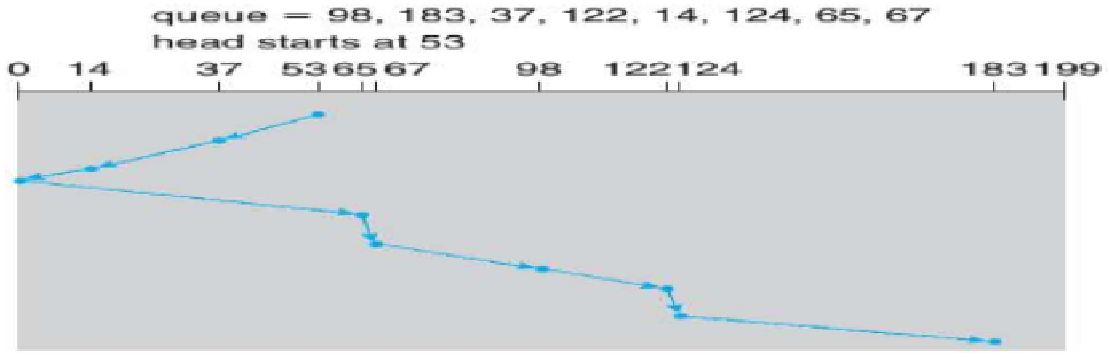
SSTF Scheduling Shortest-Seek-Time-First (SSTF)
algorithm service all the requests close to the current head position before moving the head far away to service other requests.
The SSTF algorithm selects the request with the least seek time from the current head position. (i.e.) SSTF chooses the pending request closest to the current head position.
Consider a disk queue with requests for I/O to blocks on cylinders in the order: 98, 183, 37, 122, 14, 124, 65, 67 The disk head is initially at cylinder 53.



Closest request to the initial head position 53 is at cylinder 65 takes 12 cylinders movements.
 Once we are at cylinder 65, the next closest request is at cylinder 67 (2 moves).

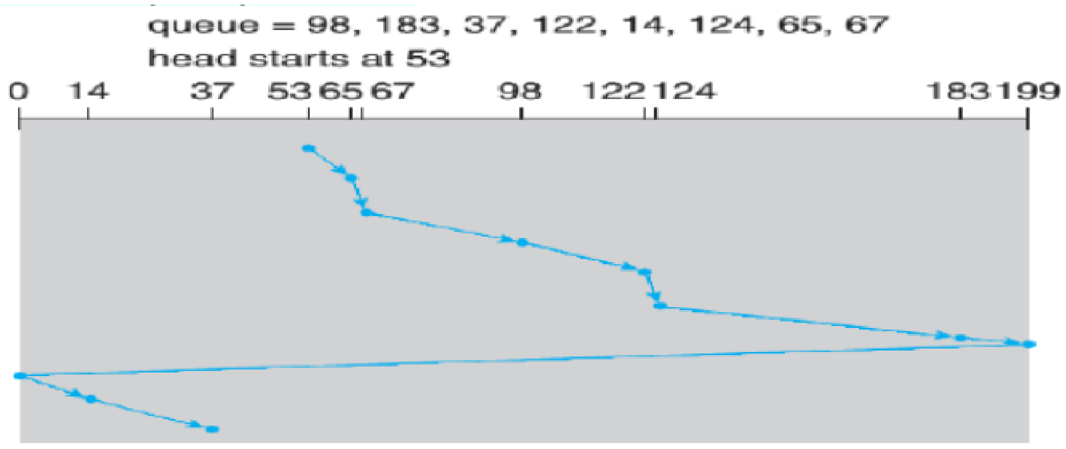
 From 67, the request at cylinder 37 is closer than the one at 98, so 37 is served next.
 Similarly we service the request at cylinder 14, then 98, 122, 124 and finally 183.
 This scheduling method results in a total head movement of only 236 cylinders. The performance of SSTF is better than FCFS but SSTF causes starvation of some requests.
 Suppose that we have two requests in the queue, for cylinders 14 and 186 and while the request from 14 is being serviced, a new request 30 near 14 arrives.
 This new request 30 will be serviced next, making the request at 186 wait.
 While request 30 is being serviced, another request close to 30 could arrive.
 A continual stream of requests near one another could cause the request for cylinder 186 to wait indefinitely.

SCAN algorithm In the SCAN algorithm,
the disk arm starts at one end of the disk and moves toward the other end, servicing requests as it reaches each cylinder, until it gets to the other end of the disk. At the other end, the direction of head movement is reversed and servicing continues. The head continuously scans back and forth across the disk. The SCAN algorithm is also called as the Elevator algorithm, since the disk arm behaves just like an elevator in a building, first servicing all the requests going up and then reversing to service requests the other way. Consider a disk queue with requests for I/O to blocks on cylinders in the order: 98, 183, 37, 122, 14, 124, 65, 67 The disk head is initially at cylinder 53.



Before applying SCAN algorithm we need to know the the direction of head movement in addition to the head’s current position.
 Assuming that the disk arm is moving toward 0 and that the initial head position is again 53 the head will next service 37 and then 14.
 At cylinder 0, the arm will reverse and will move toward the other end of the disk, servicing the requests at 65, 67, 98, 122, 124 and 183.
 A request arrives in the queue in front of the head, it will be serviced almost immediately.
 A request arriving just behind the head will have to wait until the arm moves to the end of the disk, reverses direction and comes back.
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 C-SCAN scheduling algorithm essentially treats the cylinders as a circular list that wraps around from the final cylinder to the first one. Consider a disk queue with requests for I/O to blocks on

cylinders in the order: 98, 183, 37, 122, 14, 124, 65, 67 The disk head is initially at cylinder 53.



REDUNDANT ARRAYS OF INDEPENDENT DISKS (RAID) STRUCTURE

 Disk drives have continued to get smaller and cheaper, so it is now economically feasible to attach many disks to a computer system.

 Having a large number of disks in a system and if they are operated in parallel we can improve the rate at which data can be read or written.

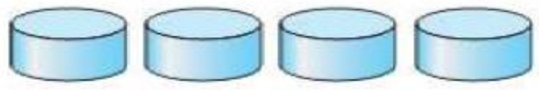
 This setup offers the potential for improving the reliability of data storage, because redundant information can be stored on multiple disks. Thus, failure of one disk does not lead to loss of data.

 This disk-organization techniques is called as Redundant Arrays Of Independent Disks (RAID).

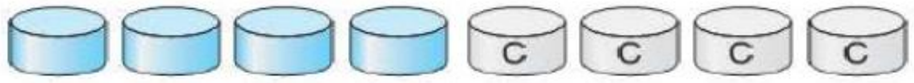
 RAIDs are used for Higher reliability and Higher data-transfer rates. RAID Levels RAIDs can be implemented in different levels:

1. RAID 0: Non-Redundant Striping
2. RAID 1: Mirrored Disks.
3. RAID 2: Memory-Style Error-Correcting Codes.
4. RAID 3: Bit-Interleaved Parity.
5. RAID 4: Block-Interleaved Parity.
6. RAID 5: Block-Interleaved Distributed Parity
7. RAID 6: P+Q redundancy 8. RAID 0 + 1 and 1 + 0

RAID 0: Non-Redundant Striping RAID level 0 refers to disk arrays with striping at the level of blocks but without any redundancy. (i.e.) some part of the data is stored in one disk other part of the data is stored in other disks without duplicating the data.


(a) RAID 0: non-redundant striping.

RAID level 1: Disk Mirroring The entire data in the disk is copied in to other disks. (i.e.) the data that is stored in one disk the same data will be copied in other disk. If one disk fails we can recover the data from its copied disk called as Backup disk.



(b) RAID 1: mirrored disks.

Note: Mirroring provides high reliability, but it is expensive. Striping provides high data-transfer rates, but it does not improve reliability.

RAID level 2. Memory-Style Error-Correctingcode (ECC) Organization

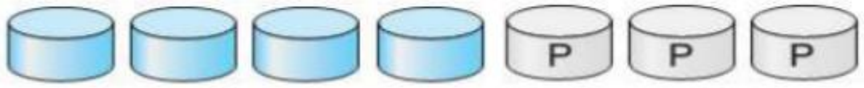
 It uses parity bits to detect errors. Each byte in a memory system may have a parity bit associated with it that records whether the number of bits in the byte set to 1 is even (parity = 0) or odd (parity = 1).

 If one of the bits in the byte is damaged (i.e.) either a 1 becomes a 0 or a 0 becomes a 1, the parity of the byte changes and thus does not match the stored parity.

 Similarly, if the stored parity bit is damaged, it does not match the computed parity.

 Thus, all single-bit errors are detected by the memory system. ECC can be used directly in disk arrays via striping of bytes across disks.

Example: The first bit of each byte can be stored in disk 1, the second bit in disk 2 and so on until the eighth bit is stored in disk 8; the error-correction bits are stored in further disks. If one of the disks fails, the remaining bits of the byte and the associated error-correction bits can be read from other disks and used to reconstruct the damaged data. RAID level 2 requires only three disks whereas RAID1 requires four disks.



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

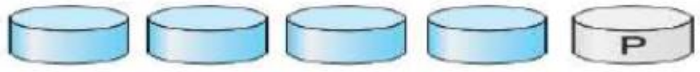
RAID level 3: Bit-Interleaved Parity

 Here, the 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.

 If one of the sectors is damaged, we know exactly which sector it is and we can figure out whether any bit in the sector is a 1 or a 0 by computing the parity of the corresponding bits from sectors in the other disks.

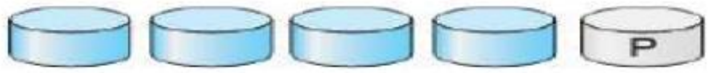
 If the parity of the remaining bits is equal to the stored parity, the missing bit is 0; otherwise, it is 1.

 RAID level 3 is less expensive than RAID2, it requires only one extra disk.



(d) RAID 3: bit-interleaved parity.

RAID level 4: Block-Interleaved Parity Organization It Uses block-level striping and keeps a parity block on a separate disk for corresponding blocks from N other disks. 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.



(e) RAID 4: block-interleaved parity.

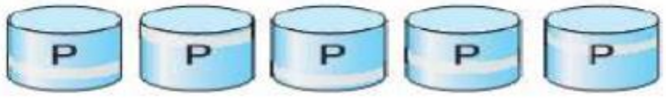
RAID level 5: Block-Interleaved Distributed Parity It differs from level 4 in that it spreads data and parity among all N+ 1 disks, rather than storing data in N disks and parity in one disk. For each block, one of the disks stores the parity and the others store data. Ex: With an array of five disks, the parity for the nth block is stored in disk (n mod 5)+1.

 The nth blocks of the other four disks store actual data for that block.

 A parity block cannot store parity for blocks in the same disk, because a disk failure would result in loss of data as well as of parity and the loss would not be recoverable.

 By spreading the parity across all the disks in the set, RAID 5 avoids potential overuse of a single parity disk, which can occur with RAID 4.

 RAID 5 is the most common parity RAID system.



(f) RAID 5: block-interleaved distributed parity.

DISK ATTACHMENT

Computers access disk storage in two ways.

1. Host-Attached Storage (HAS)
2. Network-Attached Storage (NAS)

Host-Attached Storage

 Host-attached storage is storage accessed through local I/O ports.

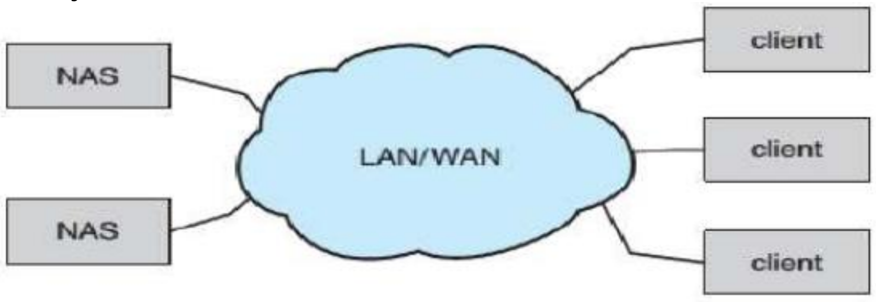
 The typical desktop PC uses an I/O bus architecture called IDE or ATA or SATA.

 This architecture supports a maximum of two drives per I/O bus.

 Hard disk drives, RAID arrays and CD, DVD and tape drives are storage devices that are suitable for use as Host-Attached Storage.

 The I/O commands that initiate data transfers to a host-attached storage device are reads and writes of logical data blocks directed to specifically identified storage units. Network-Attached Storage

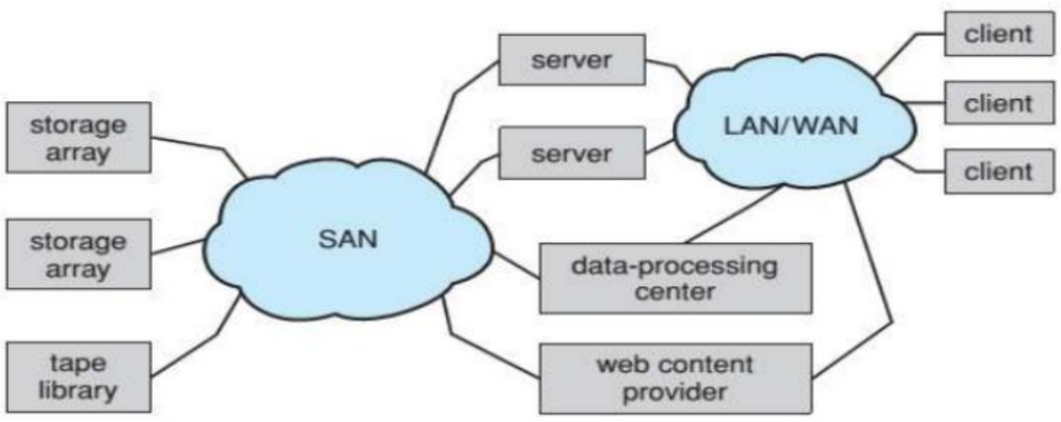
 A network-attached storage (NAS) device is a special-purpose storage system that is accessed remotely over a data network.



 Clients access network-attached storage via a remote-procedure-call interface such as NFS for UNIX systems or CIFS for Windows machines.

 The remote procedure calls (RPCs) are carried via TCP or UDP over an IP network usually the same local-area network (LAN) that carries all data traffic to the clients.

 Network-attached storage provides a convenient way for all the computers on a LAN to share a pool of storage with the same ease of naming and access enjoyed with local host-attached storage. Storage-Area Network A storage-area network (SAN) is a private network connecting servers and storage units.



 Multiple hosts and multiple storage arrays can attach to the same SAN and storage can be dynamically allocated to hosts. A SAN switch allows or prohibits access between the hosts and the storage.   Example: If a host is running low on disk space, the SAN can be configured to allocate more storage to that host. SANs make it possible for clusters of servers to share the same storage and for storage arrays to include multiple direct host connections.