# SATYAPRAKASH SWAIN

**ASSISTANT PROFESSOR, IMIT, CUTTACK.**


**LECTURE NOTE**

**OPERATING SYSTEM**

**MCA 2nd SEMESTER**

# OPERATING SYSTEM

# LECTURE NOTE

# MODULE-1

## Operating System Introduction-

An operating system acts as an intermediary between the user of a computer and computer hardware. The purpose of an operating system is to provide an environment in which a user can execute programs in a convenient and efficient manner.

An operating system is a software that manages the computer hardware. The hardware must provide appropriate mechanisms to ensure the correct operation of the computer system and to prevent user programs from interfering with the proper operation of the system.

Some examples of operating systems are UNIX, Mach, MS-DOS, MS-Windows, Windows/NT, Chicago, OS/2, MacOS, VMS, MVS, and VM.

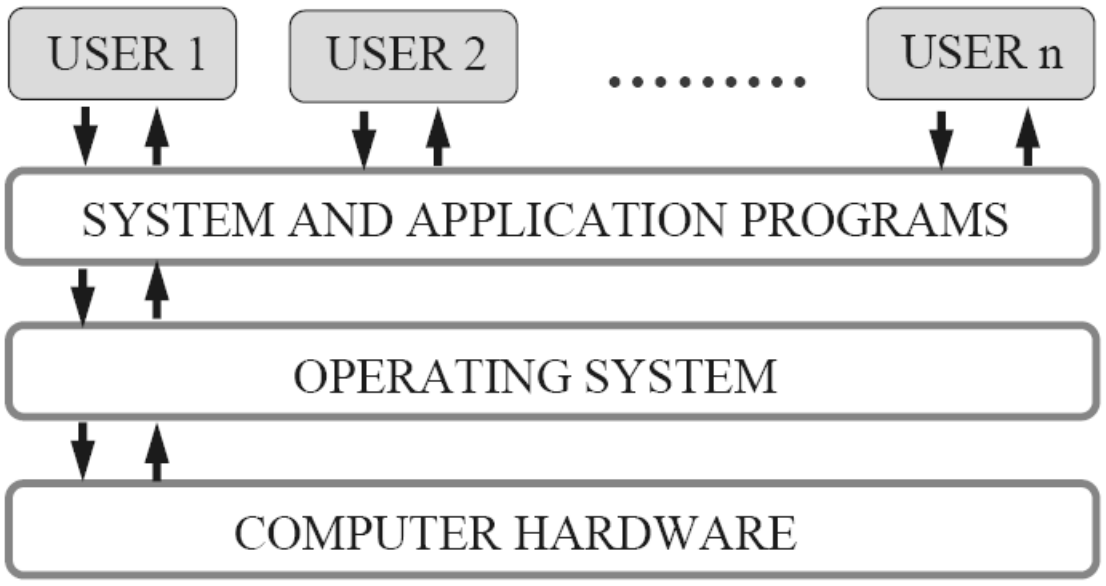
**Operating system goals:**

- ➔ Execute user programs and make solving user problems easier.
- ➔ Make the computer system convenient to use.
- ➔ Use the computer hardware in an efficient manner.

**Computer System Components**

- ➔ Hardware – provides basic computing resources (CPU, memory, I/O devices).
- ➔ Operating system – controls and coordinates the use of the hardware among the various application programs for the various users.
- ➔ Applications programs – Define the ways in which the system resources are used to solve the computing problems of the users (compilers, database systems, video games, business programs).
- ➔ Users (people, machines, other computers).

**Abstract View of System Components**




**Operating System – Definition:**

- ➔ An operating system is a program that controls the execution of application programs and acts as an interface between the user of a computer and the computer hardware.
- ➔ A more common definition is that the operating system is the one program running at all times on the computer (usually called the kernel), with all else being application programs.
- ➔ An operating system is concerned with the allocation of resources and services, such as memory, processors, devices, and information. The operating system correspondingly includes programs to manage these resources, such as a traffic controller, a scheduler, memory management module, I/O programs, and a file system.

## Functions of Operating System:

1. **Process Management**

A process is a program in execution. A process needs certain resources, including CPU time, memory, files, and I/O devices, to accomplish its task.

The operating system is responsible for the following activities in connection with process management.

- Process creation and deletion.
- process suspension and resumption.
- Provision of mechanisms for:
    i. process synchronization
    ii. process communication

2. **Main-Memory Management**

Memory is a large array of words or bytes, each with its own address. It is a repository of quickly accessible data shared by the CPU and I/O devices.

Main memory is a volatile storage device. It loses its contents in the case of system failure.

The operating system is responsible for the following activities in connections with memory management:

- Keep track of which parts of memory are currently being used and by whom.
- Decide which processes to load when memory space becomes available.
- Allocate and de-allocate memory space as needed.

3. **File Management**

A file is a collection of related information defined by its creator. Commonly, files represent programs (both source and object forms) and data.

The operating system is responsible for the following activities in connections with file management:

- File creation and deletion.
- Directory creation and deletion.
- Support of primitives for manipulating files and directories.
- Mapping files onto secondary storage.
- File backup on stable (non-volatile) storage media.

4. **I/O System Management**

The I/O system consists of:

- A buffer-caching system
- A general device-driver interface
- Drivers for specific hardware devices

5. **Secondary Storage Management**

Since main memory (primary storage) is volatile and too small to accommodate all data and programs permanently, the computer system must provide secondary storage to back up main memory.

Most modern computer systems use disks as the principle on-line storage medium, for both programs and data.

The operating system is responsible for the following activities in connection with disk management:

- Free space management
- Storage allocation
- Disk scheduling

6. **Protection System**

- Protection refers to a mechanism for controlling access by programs, processes, or users to both system and user resources.
- The protection mechanism must:
    a. distinguish between authorized and unauthorized usage.
    b. specify the controls to be imposed.
    c. provide a means of enforcement.

## Characteristics:

Here is a list of some of the most prominent characteristic features of Operating Systems −

- **Memory Management-** Keeps track of the primary memory, i.e. what part of it is in use by whom, what part is not in use, etc. and allocates the memory when a process or program requests it.
- **Processor Management-** Allocates the processor (CPU) to a process and de-allocates the processor when it is no longer required.
- **Device Management-** Keeps track of all the devices. This is also called I/O controller that decides which process gets the device, when, and for how much time.
- **File Management-** Allocates and de-allocates the resources and decides who gets the resources.
- **Security-** Prevents unauthorized access to programs and data by means of passwords and other similar techniques.
- **Job Accounting-** Keeps track of time and resources used by various jobs and/or users.
- **Control Over System Performance-** Records delays between the request for a service and from the system.
- **Interaction with the Operators-** Interaction may take place via the console of the computer in the form of instructions. The Operating System acknowledges the same, does the corresponding action, and informs the operation by a display screen.
- **Error-detecting Aids -** Production of dumps, traces, error messages, and other debugging and error-detecting methods.
- **Coordination Between Other Software and Users-** Coordination and assignment of compilers, interpreters, assemblers, and other software to the various users of the computer systems.

## Evolution of OS:

### 1. Batch Processing Operating System:

- This type of OS accepts more than one jobs and these jobs are batched/ grouped together according to their similar requirements. This is done by computer operator. Whenever the computer becomes available, the batched jobs are sent for execution and gradually the output is sent back to the user.
- It allowed only one program at a time.
- This OS is responsible for scheduling the jobs according to priority and the resource required.

### 2. Multiprogramming Operating System:

- This type of OS is used to execute more than one jobs simultaneously by a single processor. it increases CPU utilization by organizing jobs so that the CPU always has one job to execute.
- The concept of multiprogramming is described as follows:
    - All the jobs that enter the system are stored in the job pool( in disc). The operating system loads a set of jobs from job pool into main memory and begins to execute.
    - During execution, the job may have to wait for some task, such as an I/O operation, to complete. In a multiprogramming system, the operating system simply switches to another job and executes. When that job needs to wait, the CPU is switched to another job, and so on.
    - When the first job finishes waiting and it gets the CPU back.
    - As long as at least one job needs to execute, the CPU is never idle.

Multiprogramming operating systems use the mechanism of job scheduling and CPU scheduling.

### 3. Time-Sharing/multitasking Operating Systems

Time sharing (or multitasking) OS is a logical extension of multiprogramming. It provides extra facilities such as:

- Faster switching between multiple jobs to make processing faster.
- Allows multiple users to share computer system simultaneously.
- The users can interact with each job while it is running.

These systems use a concept of virtual memory for effective utilization of memory space. Hence, in this OS, no jobs are discarded. Each one is executed using virtual memory concept. It uses CPU scheduling, memory management, disc management and security management. Examples: CTSS, MULTICS, CAL, UNIX etc.

**4. Multiprocessor Operating Systems**

Multiprocessor operating systems are also known as parallel OS or tightly coupled OS. Such operating systems have more than one processor in close communication that sharing the computer bus, the clock and sometimes memory and peripheral devices. It executes multiple jobs at same time and makes the processing faster.

**Multiprocessor systems have three main advantages**:

- **Increased throughput:** By increasing the number of processors, the system performs more work in less time. The speed-up ratio with N processors is less than N.
- **Economy of scale:** Multiprocessor systems can save more money than multiple single-processor systems, because they can share peripherals, mass storage, and power supplies.
- **Increased reliability:** If one processor fails to done its task, then each of the remaining processors must pick up a share of the work of the failed processor. The failure of one processor will not halt the system, only slow it down.

The ability to continue providing service proportional to the level of surviving hardware is called graceful degradation. Systems designed for graceful degradation are called fault tolerant.

The multiprocessor operating systems are classified into two categories:

- Symmetric multiprocessing system
- Asymmetric multiprocessing system

➔ **In symmetric multiprocessing system**, each processor runs an identical copy of the operating system, and these copies communicate with one another as needed.

➔ **In asymmetric multiprocessing system**, a processor is called master processor that controls other processors called slave processor. Thus, it establishes master-slave relationship. The master processor schedules the jobs and manages the memory for entire system.

**5. Distributed Operating Systems**

- In distributed system, the different machines are connected in a network and each machine has its own processor and own local memory.
- In this system, the operating systems on all the machines work together to manage the collective network resource.
- It can be classified into two categories:
  - Client-Server systems
  - Peer-to-Peer systems

Advantages of distributed systems.

- Resources Sharing
- Computation speed up – load sharing
- Reliability
- Communications
- Requires networking infrastructure.
- Local area networks (LAN) or Wide area networks (WAN)

.

**6. Desktop Systems/Personal Computer Systems**

➔ The PC operating system is designed for maximizing user convenience and responsiveness. This system is neither multi-user nor multitasking.

➔ These systems include PCs running Microsoft Windows and the Apple Macintosh. The MS-DOS operating system from Microsoft has been superseded by multiple flavors of Microsoft Windows and IBM has upgraded MS-DOS to the OS/2 multitasking system.

➔ The Apple Macintosh operating system has been ported to more advanced hardware, and now includes new features such as virtual memory and multitasking.

**7. Real-Time Operating Systems (RTOS)**

➔ A real-time operating system (RTOS) is a multitasking operating system intended for applications with fixed deadlines (real-time computing). Such applications include some small embedded systems, automobile engine controllers, industrial robots, spacecraft, industrial control, and some large-scale computing systems.

➔ The real time operating system can be classified into two categories:

1. hard real time system and
2. soft real time system.

➔ **A hard real-time system** guarantees that critical tasks be completed on time. This goal requires that all delays in the system be bounded, from the retrieval of stored data to the time that it takes the operating system to finish any request made of it. Such time constraints dictate the facilities that are available in hard real-time systems.

➔ **A soft real-time system** is a less restrictive type of real-time system. Here, a critical real-time task gets priority over other tasks and retains that priority until it completes. Soft real time system can be mixed with other types of systems. Due to less restriction, they are risky to use for industrial control and robotics.

## Operating System Structures

- System Components
- Operating System Services
- System Calls
- System Programs
- System Structure
- Virtual Machines
- System Design and Implementation
- System Generation

## Common System Components

- Process Management
- Main Memory Management
- File Management
- I/O System Management
- Secondary Management
- Networking
- Protection System
- Command-Interpreter System

## Operating System Services:

Following are the five services provided by operating systems to the convenience of the users.

**➔Program Execution**

The purpose of computer systems is to allow the user to execute programs. So the operating system provides an environment where the user can conveniently run programs. Running a program involves the allocating and deallocating memory, CPU scheduling in case of multiprocessing.

**➔I/O Operations**

Each program requires an input and produces output. This involves the use of I/O. So the operating systems are providing I/O makes it convenient for the users to run programs.

**➔File System Manipulation**

The output of a program may need to be written into new files or input taken from some files. The operating system provides this service.

➔**Communications**

The processes need to communicate with each other to exchange information during execution. It may be between processes running on the same computer or running on the different computers. Communications can be occur in two ways: (i) shared memory or (ii) message passing

➔**Error Detection**

An error is one part of the system may cause malfunctioning of the complete system. To avoid such a situation operating system constantly monitors the system for detecting the errors. This relieves the user of the worry of errors propagating to various part of the system and causing malfunctioning.

Following are the three services provided by operating systems for ensuring the efficient operation of the system itself.

➔**Resource allocation**

When multiple users are logged on the system or multiple jobs are running at the same time, resources must be allocated to each of them. Many different types of resources are managed by the operating system.

➔**Accounting**

The operating systems keep track of which users use how many and which kinds of computer resources. This record keeping may be used for accounting (so that users can be billed) or simply for accumulating usage statistics.

➔**Protection**

When several disjointed processes execute concurrently, it should not be possible for one process to interfere with the others, or with the operating system itself. Protection involves ensuring that all access to system resources is controlled. Security of the system from outsiders is also important. Such security starts with each user having to authenticate him to the system, usually by means of a password, to be allowed access to the resources

## System Call:

➔System calls provide an interface between the process and the operating system.

➔System calls allow user-level processes to request some services from the operating system which process itself is not allowed to do.

➔For example, for I/O a process involves a system call telling the operating system to read or write particular area and this request is satisfied by the operating system.

The following different types of system calls provided by an operating system:

**Process control**

- end, abort
- load, execute
- create process, terminate process
- get process attributes, set process attributes
- wait for time
- wait event, signal event
- allocate and free memory

**File management**

- create file, delete file
- open, close

- read, write, reposition
- get file attributes, set file attributes

**Device management**

- request device, release device
- read, write, reposition
- get device attributes, set device attributes
- logically attach or detach devices

**Information maintenance**

- get time or date, set time or date
- get system data, set system data
- get process, file, or device attributes
- set process, file, or device attributes

**Communications**

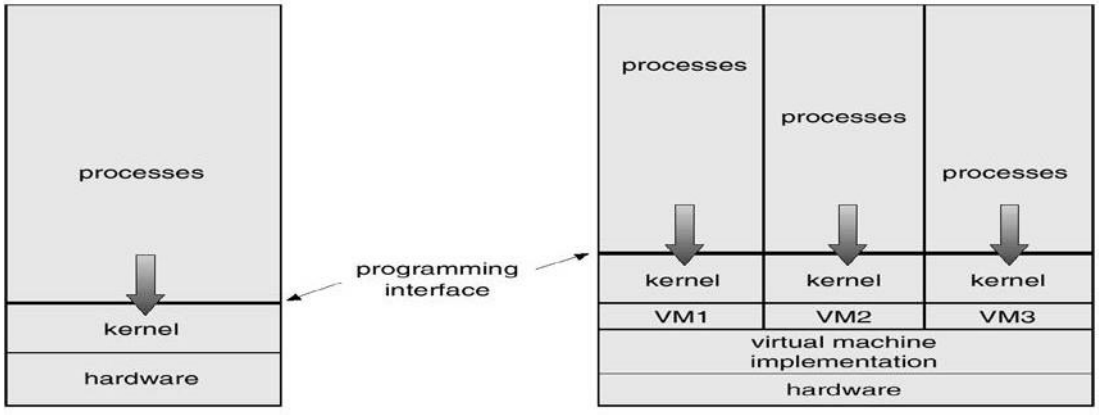
- create, delete communication connection
- send, receive messages
- transfer status information
- attach or detach remote devices

## Virtual Machines

- ➔ A virtual machine takes the layered approach to its logical conclusion. It treats hardware and the operating system kernel as though they were all hardware.
- ➔ A virtual machine provides an interface identical to the underlying bare hardware.
- ➔ The operating system creates the illusion of multiple processes, each executing on its own processor with its own (virtual) memory.
- ➔ The resources of the physical computer are shared to create the virtual machines.

- CPU scheduling can create the appearance that users have their own processor.
- Spooling and a file system can provide virtual card readers and virtual line printers.
- A normal user time-sharing terminal serves as the virtual machine operator's console.

## System Models




Non-virtual Machine Virtual Machine

**Advantages/Disadvantages of Virtual Machines**

- The virtual-machine concept provides complete protection of system resources since each virtual machine is isolated from all other virtual machines. This isolation, however, permits no direct sharing of resources.
- A virtual-machine system is a perfect vehicle for operating-systems research and development. System development is done on the virtual machine, instead of on a physical machine and so does not disrupt normal system operation.
- The virtual machine concept is difficult to implement due to the effort required to provide an exact duplicate to the underlying machine.

**Process and CPU Scheduling**

Class Note.

# MODULE-2

**Memory Management and Virtual Memory** –
Class Note.

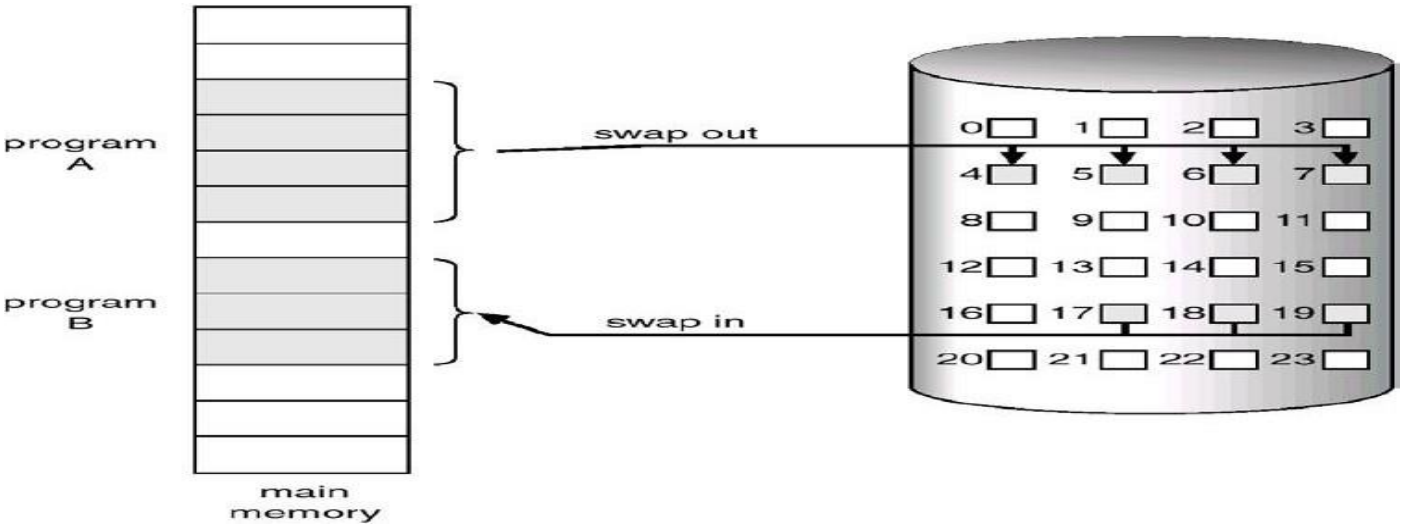
**Demand Paging:**

A demand-paging system is similar to a paging system with swapping. Generally, Processes reside on secondary memory (which is usually a disk). When we want to execute a process, we swap it into memory. Rather than swapping the entire process into memory, it swaps the required page. This can be done by a **lazy swapper.**

A lazy swapper never swaps a page into memory unless that page will be needed. A swapper manipulates entire processes, whereas a **pager** is concerned with the individual pages of a process.

## Page transfer Method:

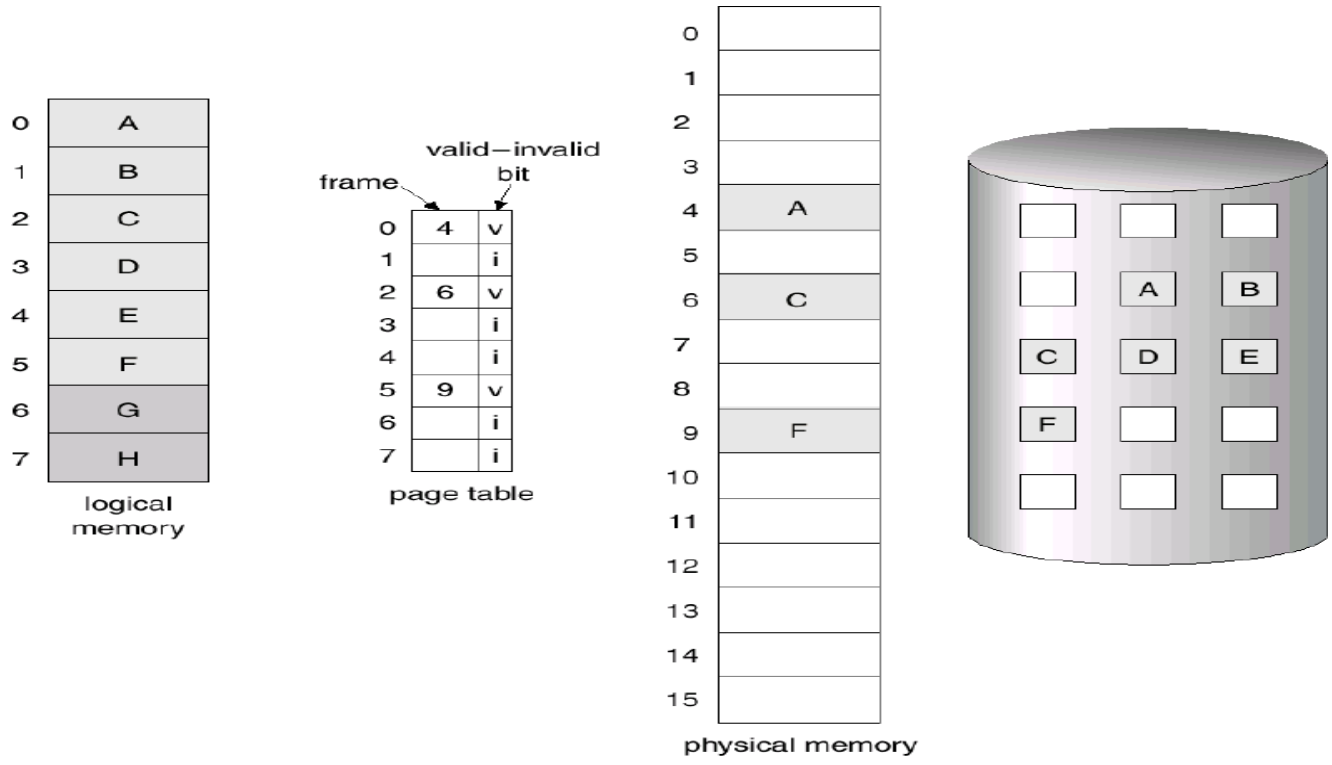
When a process is to be swapped in, the pager guesses which pages will be used before the process is swapped out again. Instead of swapping in a whole process, the pager brings only those necessary pages into memory. Thus, it avoids reading into memory pages that will not be used anyway, decreasing the swap time and the amount of physical memory needed.




(Transfer of a paged memory to contiguous disk space)

**Page Table:**

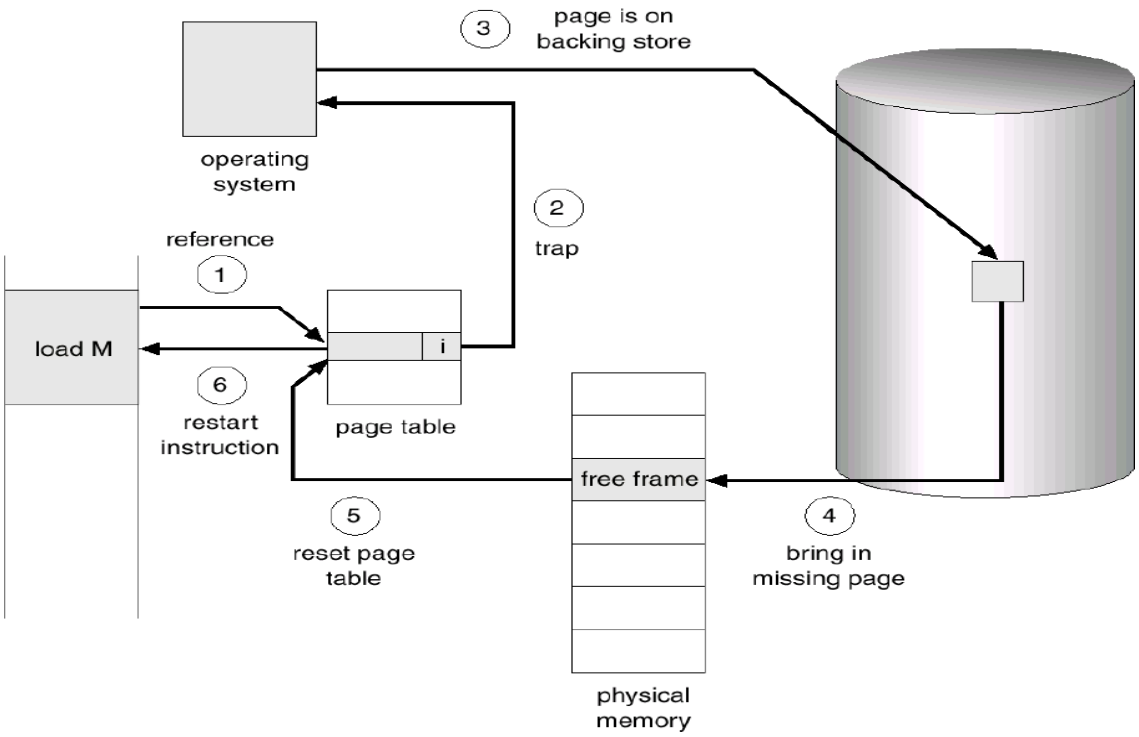
- The valid-invalid bit scheme of Page table can be used for indicating which pages are currently in memory.
- When this bit is set to "valid", this value indicates that the associated page is both legal and in memory. If the bit is set to "invalid", this value indicates that the page either is not valid or is valid but is currently on the disk.
- The page-table entry for a page that is brought into memory is set as usual, but the page- table entry for a page that is not currently in memory is simply marked invalid, or contains the address of the page on disk.




(Page table when some pages are not in main memory)

When a page references an invalid page, then it is called **Page Fault**. It means that page is not in main memory. The procedure for handling page fault is as follows:

1. We check an internal table for this process, to determine whether the reference was a valid or invalid memory access.
2. If the reference was invalid, we terminate the process. If it was valid, but we have not yet brought in that page in to memory.
3. We find a free frame (by taking one from the free-frame list).
4. We schedule a disk operation to read the desired page into the newly allocated frame.
5. When the disk read is complete, we modify the internal table kept with the process and the page table to indicate that the page is now in memory.
6. We restart the instruction that was interrupted by the illegal address trap. The process can now access the page as though it had always been in memory.




(Diagram of Steps in handling a page fault)

**Note:** The pages are copied into memory, only when they are required. This mechanism is called **Pure Demand Paging**.

## Performance of Demand Paging

Let p be the probability of a page fault ($0 \leq p \leq 1$). Then the **effective access time** is

**Effective access time = (1 - p) x memory access time + p x page fault time**

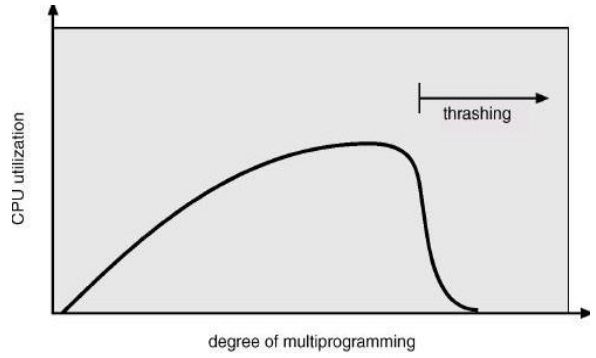
In any case, we are faced with three major components of the page-fault service time:

1. Service the page-fault interrupt.
2. Read in the page.
3. Restart the process.

**Thrashing:**

The system spends most of its time shuttling pages between main memory and secondary memory due to frequent page faults. This behavior is known as thrashing.

A process is thrashing if it is spending more time paging than executing. This leads to: low CPU utilization and the operating system thinks that it needs to increase the degree of multiprogramming.

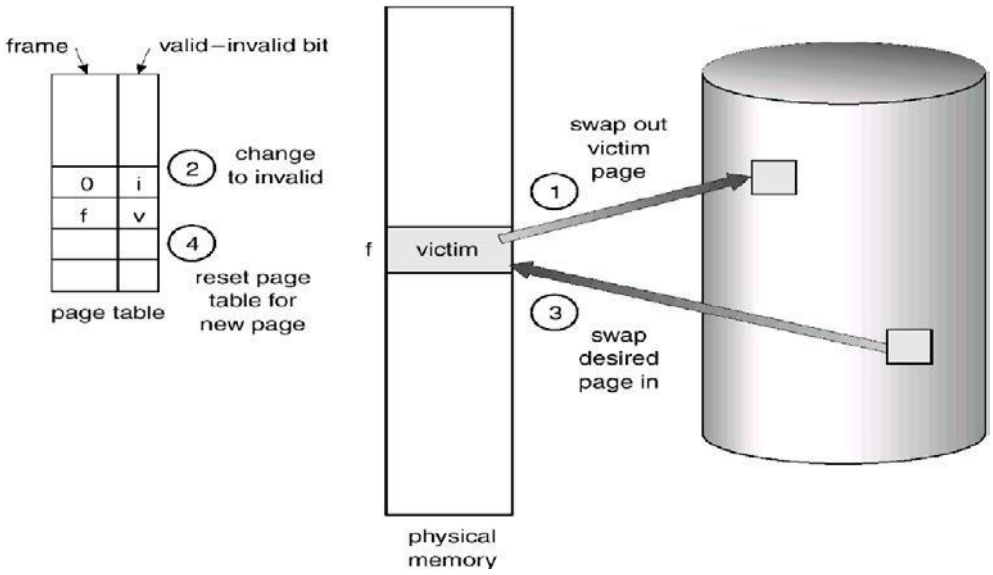



(Thrashing)

**Page Replacement,**

**The page replacement** is a mechanism that loads a page from disc to memory when a page of memory needs to be allocated. Page replacement can be described as follows:

1. Find the location of the desired page on the disk.
2. Find a free frame:
    a. If there is a free frame, use it.
    b. If there is no free frame, use a page-replacement algorithm to select a victim frame.
    c. Write the victim page to the disk; change the page and frame tables accordingly.
3. Read the desired page into the (newly) free frame; change the page and frame tables.
4. Restart the user process.




(Diagram of Page replacement)

**Page Replacement Algorithm:**

Examples of text book.

**Allocation of Frames:**

When a page fault occurs, there is a free frame available to store new page into a frame. While the page swap is taking place, a replacement can be selected, which is written to the disk as the user process continues to execute. The operating system allocate all its buffer and table space from the free-frame list for new page.

Two major allocation Algorithm/schemes.

1. **equal allocation**
2. **proportional allocation**

**1. Equal allocation:** The easiest way to split m frames among *n* processes is to give everyone an equal share, m/n frames. This scheme is called **equal allocation.**

**2. proportional allocation:** Here, it allocates available memory to each process according to its size. Let the size of the virtual memory for process $p_i$ be $s_i$, and define $S = \sum Si$

Then, if the total number of available frames is *m*, we allocate $a_i$ frames to process $p_i$, where $a_i$ is approximately $a_i = Si/S \times m$.

# MODULE-3

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

## FILE CONCEPTS:

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 cannot 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. First, space in the file system must be found for the file. 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:

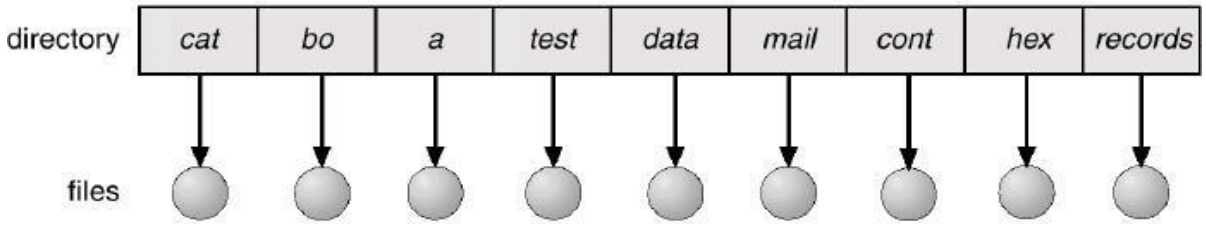
**Sequential Access:** Information in the file is processed in order, one record after the other. A read operation reads the next portion of the file and automatically advances a file pointer, which tracks the I/O location. Similarly, a write appends to the end of the file and advances to the end of 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 modified to include the block number as a parameter. Thus, we have *read n*, where *n* is the block number, rather than *read next*, and *write n* rather than *write next*.

## 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: Single-level tree, Two-level tree, multi-level tree or cyclic graph.

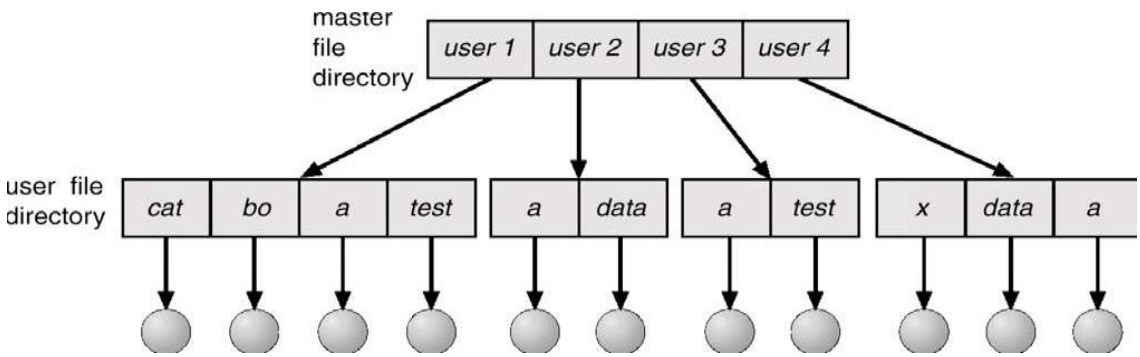
**Single-Level Directory:** The simplest directory structure is the single-level directory. All files are contained in the same directory, which is easy to support and understand. A single-level directory has significant limitations, when the number of files increases or when the system has more than one user. Since all files are in the same directory, they must have unique names.



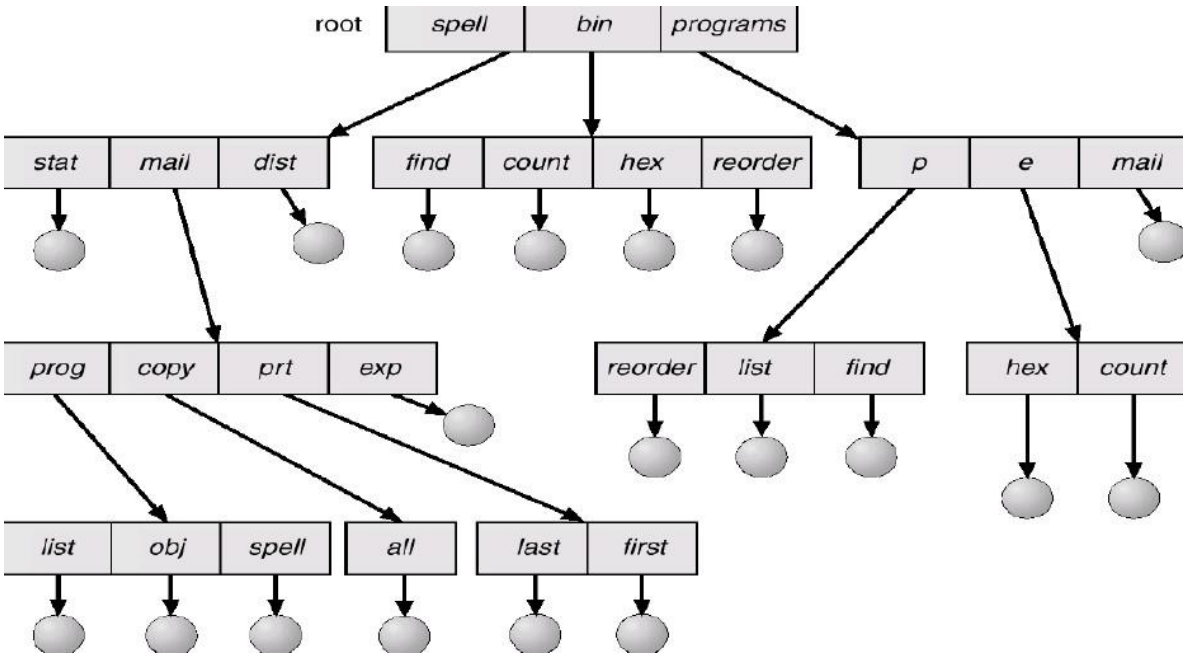

**Two-Level Directory:** In the two-level directory structure, each user has its own **user file directory (UFD).** Each UFD has a similar structure, but lists only the files of a single user. When a user job starts or a user logs in, the system's **master file directory (MFD)** is searched. The MFD 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.



**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 sub- directories 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.

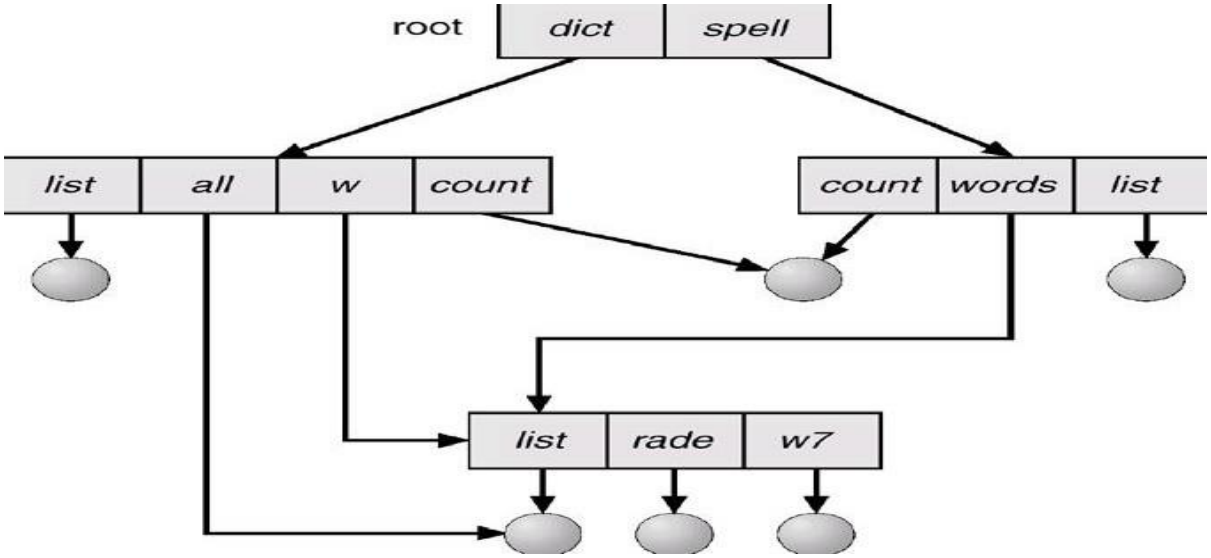



**Acyclic-Graph Directories:**

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.

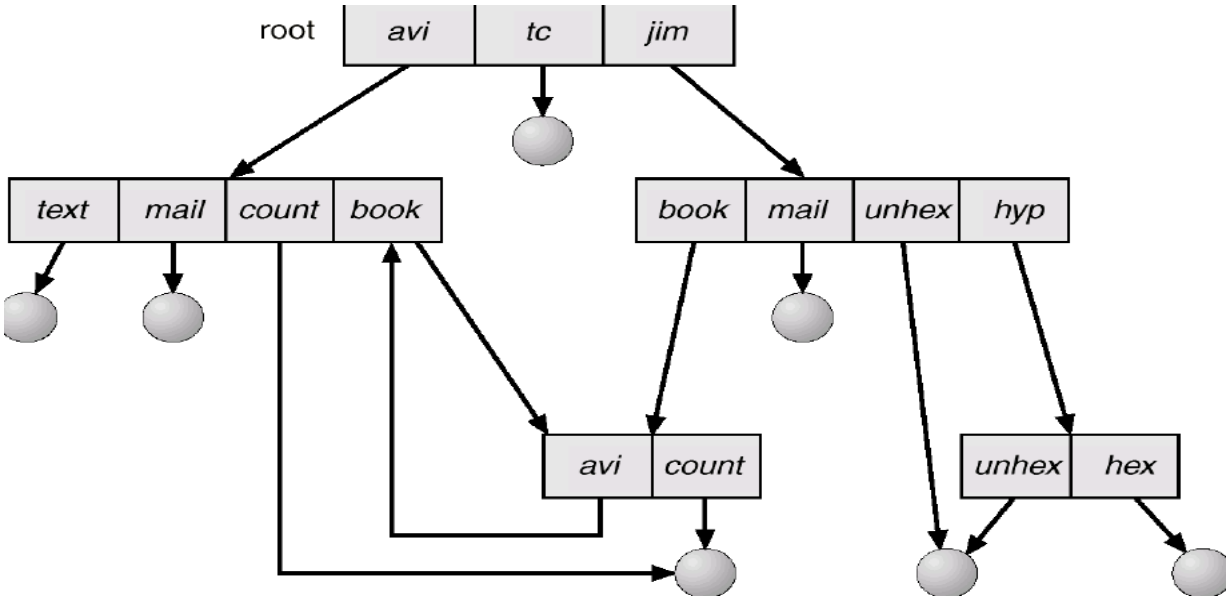
A shared file (or directory) is not the same as two copies of the file. With two copies, each programmer can view the copy rather than the original, but if one programmer changes the file, the changes will not appear in the other's copy.

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.




**General Graph Directory:**

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




## File Protection:

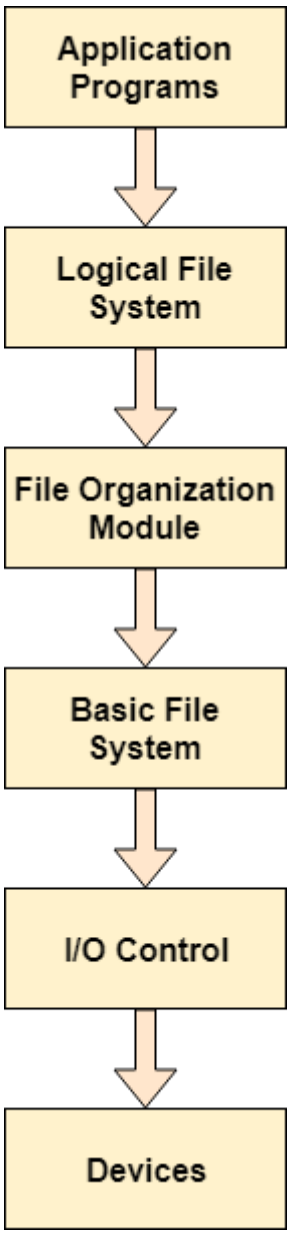
Methods used in protecting valuable data on computers. File protection is accomplished by password protecting a file or only providing rights to a specific user or group.

## File System Structure:

File System provide efficient access to the disk by allowing data to be stored, located and retrieved in a convenient way. A file System must be able to store the file, locate the file and retrieve the file.

Most of the Operating Systems use layering approach for every task including file systems. Every layer of the file system is responsible for some activities.

The image shown below, elaborates how the file system is divided in different layers, and also the functionality of each layer.




- ➔ When an application program asks for a file, the first request is directed to the logical file system. The logical file system contains the Meta data of the file and directory structure. If the application program doesn't have the required permissions of the file then this layer will throw an error. Logical file systems also verify the path to the file.
- ➔ Generally, files are divided into various logical blocks. Files are to be stored in the hard disk and to be retrieved from the hard disk. Hard disk is divided into various tracks and sectors. Therefore, in order to store and retrieve the files, the logical blocks need to be mapped to physical blocks. This mapping is done by File organization module. It is also responsible for free space management.
- ➔ Once File organization module decided which physical block the application program needs, it passes this information to basic file system. The basic file system is responsible for issuing the commands to I/O control in order to fetch those blocks.

➔ I/O controls contain the codes by using which it can access hard disk. These codes are known as device drivers. I/O controls are also responsible for handling interrupts.

## Allocation methods:

Files are allocated disk spaces by operating system. Operating systems deploy following three main ways to allocate disk space to files.

- Contiguous Allocation
- Linked Allocation
- Indexed Allocation

**Contiguous Allocation**

- Each file occupies a contiguous address space on disk.
- Assigned disk address is in linear order.
- Easy to implement.
- External fragmentation is a major issue with this type of allocation technique.

**Linked Allocation**

- Each file carries a list of links to disk blocks.
- Directory contains link / pointer to first block of a file.
- No external fragmentation
- Effectively used in sequential access file.
- Inefficient in case of direct access file.

**Indexed Allocation**

- Provides solutions to problems of contiguous and linked allocation.
- A index block is created having all pointers to files.
- Each file has its own index block which stores the addresses of disk space occupied by the file.
- Directory contains the addresses of index blocks of files.

(For Diagram, follow your text book)

## Free-space Management:

The system keeps tracks of the free disk blocks for allocating space to files when they are created. Also, to reuse the space released from deleting the files, free space management becomes crucial. The system maintains a free space list which keeps track of the disk blocks that are not allocated to some file or directory. The free space list can be implemented mainly as:

1. **Bitmap or Bit vector –**

A Bitmap or Bit Vector is series or collection of bits where each bit corresponds to a disk block. The bit can take two values: 0 and 1: *0 indicates that the block is allocated* and 1 indicates a free block.
The given instance of disk blocks on the disk in *Figure 1* (where green blocks are allocated) can be represented by a bitmap of 16 bits as: **0000111000000110**.

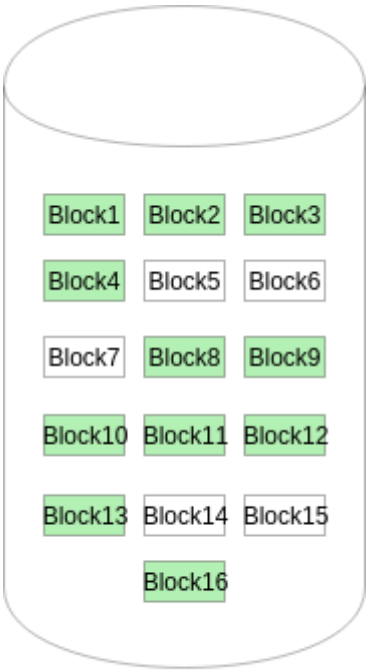



**Figure** - 1

**Advantages –**

- Simple to understand.
- Finding the first free block is efficient. It requires scanning the words (a group of 8 bits) in a bitmap for a non-zero word. (A 0-valued word has all bits 0). The first free block is then found by scanning for the first 1 bit in the non-zero word.

The block number can be calculated as:

*(number of bits per word) *(number of 0-values words) + offset of bit first bit 1 in the non-zero word* .

For the *Figure-1*, we scan the bitmap sequentially for the first non-zero word. The first group of 8 bits (00001110) constitute a non-zero word since all bits are not 0. After the non-0 word is found, we look for the first 1 bit. This is the 5th bit of the non-zero word. So, offset =5. Therefore, the first free block number = 8*0+5 = 5.

2. **Linked List –**

In this approach, the free disk blocks are linked together i.e. a free block contains a pointer to the next free block. The block number of the very first disk block is stored at a separate location on disk and is also cached in memory.

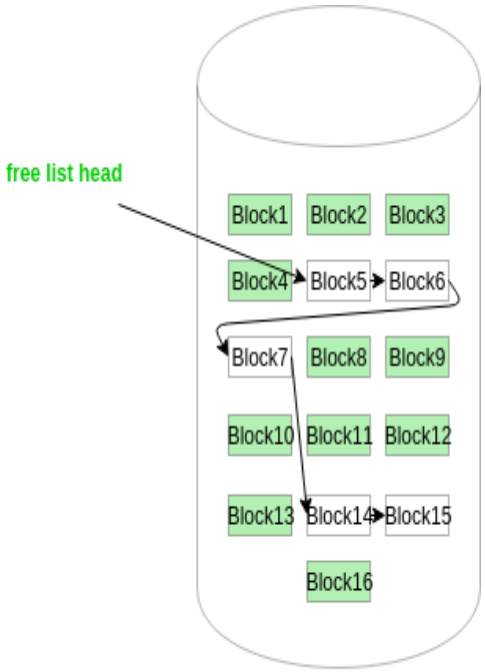



Figure - 2

In *Figure-2*, the free space list head points to Block 5 which points to Block 6, the next free block and so on. The last free block would contain a null pointer indicating the end of free list. A drawback of this method is the I/O required for free space list traversal.

➔ **Grouping** –
This approach stores the address of the free blocks in the first free block. The first free block stores the address of some, say n free blocks. Out of these n blocks, the first n-1 blocks are actually free and the last block contains the address of next free n blocks. An **advantage** of this approach is that the addresses of a group of free disk blocks can be found easily.

➔ **Counting** –
This approach stores the address of the first free disk block and a number n of free contiguous disk blocks that follow the first block.

Every entry in the list would contain:

1. Address of first free disk block
2. A number n

For example, *in Figure-1*, the first entry of the free space list would be: ([Address of Block 5], 2), because 2 contiguous free blocks follow block 5.

## Directory Management:

Directory is a place/area/location where a set of file(s) will be stored. It is a folder which contains details about files, file size and time when they are created and last modified. The different types of directories are discussed below −

**Root Directory**

Root Directory is created when we start formatting the disk and start putting files on it. In this, we can create new directories called "sub-directories". Root directory is the highest level directory and is seen when booting a system.

**Subdirectory**

Subdirectory is a directory inside root directory, in turn, it can have another sub-directory in it.

## Directory Implementation:

There is the number of algorithms by using which, the directories can be implemented. However, the selection of an appropriate directory implementation algorithm may significantly affect the performance of the system.

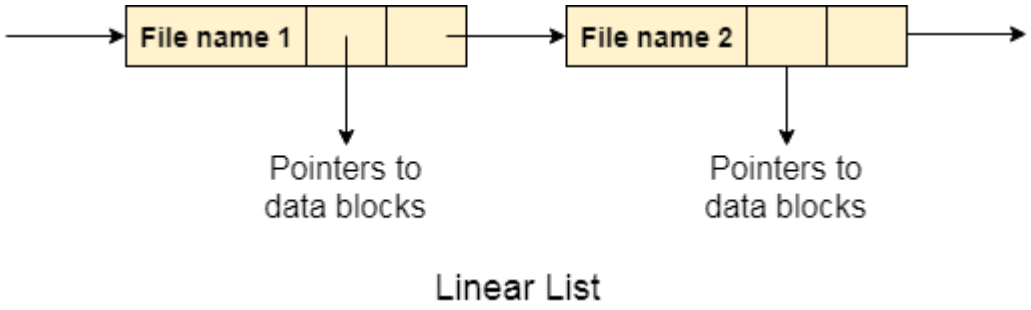
The directory implementation algorithms are classified according to the data structure they are using. There are mainly two algorithms which are used in these days.

**1. Linear List**

In this algorithm, all the files in a directory are maintained as singly lined list. Each file contains the pointers to the data blocks which are assigned to it and the next file in the directory.

**Characteristics**

1. When a new file is created, then the entire list is checked whether the new file name is matching to a existing file name or not. In case, it doesn't exist, the file can be created at the beginning or at the end. Therefore, searching for a unique name is a big concern because traversing the whole list takes time.
2. The list needs to be traversed in case of every operation (creation, deletion, updating, etc) on the files therefore the systems become inefficient.



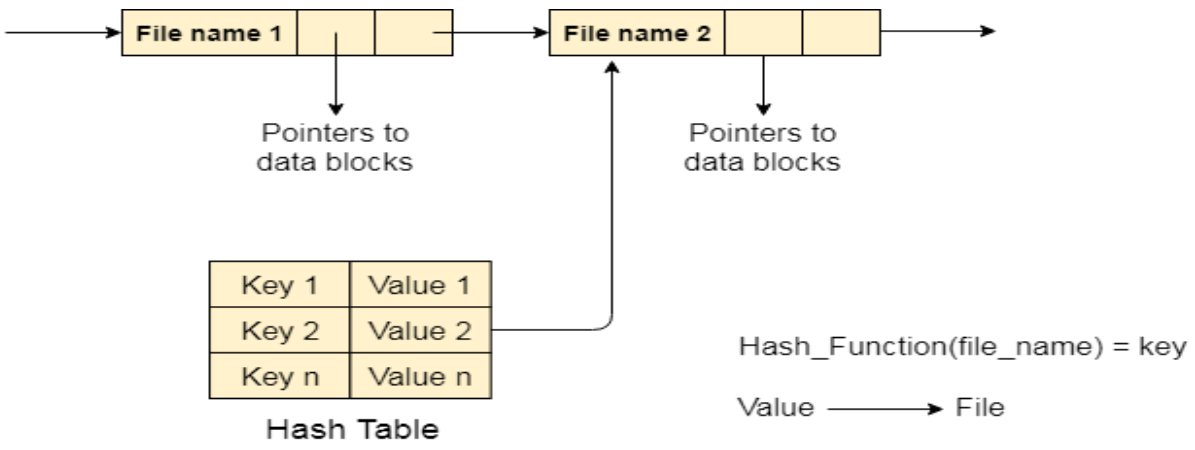

Linear List

**2. Hash Table**

To overcome the drawbacks of singly linked list implementation of directories, there is an alternative approach that is hash table. This approach suggests to use hash table along with the linked lists.

A key-value pair for each file in the directory gets generated and stored in the hash table. The key can be determined by applying the hash function on the file name while the key points to the corresponding file stored in the directory.

Now, searching becomes efficient due to the fact that now, entire list will not be searched on every operating. Only hash table entries are checked using the key and if an entry found then the corresponding file will be fetched using the value.




Hash Table

## Efficiency and Performance:

Efficiency dependent on:

- ➔ disk allocation and directory algorithms
- ➔ types of data kept in file's directory entry

Performance

- ➔ disk cache – separate section of main memory for frequently used blocks
- ➔ free-behind and read-ahead – techniques to optimize sequential access
- ➔ improve PC performance by dedicating section of memory as virtual disk or RAM disk

## The Critical Section Problem:

Consider a system consisting of n processes {Po,P1, ..., Pn-1). Each process has a segment of code, called a **critical section**, in which the process may be changing common variables, updating a table, writing a file, and so on. The important feature of the system is that, when one process is executing in its critical section, no other process is to be allowed to execute in its critical section. Thus, the execution of critical sections by the processes is **mutually exclusive** in time. The critical-section problem is to design a protocol that the processes can use to cooperate. Each process must request permission to enter its critical section. The section of code implementing this request is the entry section. The critical section may be followed by an exit section. The remaining code is the remainder section.

```
do{
        Entry section
                Critical section
        Exit section
                Remainder section
}while(1);
```

A solution to the critical-section problem must satisfy the following three requirements:

**1. Mutual Exclusion:** If process Pi is executing in its critical section, then no other processes can be executing in their critical sections.
**2. Progress:** If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder section can participate in the decision on which will enter its critical section next, and this selection cannot be postponed indefinitely.
**3. Bounded Waiting:** There exists a bound on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

## Synchronization Hardware:

As with other aspects of software, hardware features can make the programming task easier and improve system efficiency. In this section, we present some simple hardware instructions that are available on many systems, and show how they can be used effectively in solving the critical-section problem.

The definition of the TestAndSet instruction.

```
boolean TestAndSet(boolean &target)
    {
            boolean rv = target;
            target = true;
            return rv;
    }
```

The critical section problem could be solved simply in a uniprocessor environment if we could forbid interrupts to occur while a shared variable is being modified. In this manner, we could be sure that the current sequence of instructions would be allowed to execute in order without preemption. No other instructions would be run, so no unexpected modifications could be made to the shared variable.

Unfortunately, this solution is not feasible in a multiprocessor environment. Disabling interrupts on a

multiprocessor can be time-consuming, as the message is passed to all the processors. This message passing delays entry into each critical section, and system efficiency decreases. Also, consider the effect on a system's clock, if the clock is kept updated by interrupts.

Many machines therefore provide special hardware instructions that allow us either to test and modify the content of a word, or to swap the contents of two words, atomically-that is, as one uninterruptible unit. We can use these special instructions to solve the critical-section problem in a relatively simple manner. Rather than discussing one specific instruction for one specific machine, let us abstract the main concepts behind these types of instructions.

The TestAndSet instruction can be defined as shown in code. The important characteristic is that this instruction is executed atomically. Thus, if two TestAndSet instructions are executed simultaneously (each on a different CPU), they will be executed sequentially in some arbitrary order.

Mutual-exclusion implementation with **TestAndSet**

```
do{

        while(TestAndSet(lock));
                critical section
        lock=false
                Remainder section
   }while(1);

void Swap(boo1ean &a, boolean &b)
{
                        boolean temp = a;
                        a = b;
                        b = temp
}
```

If the machine supports the TestAndSet instruction, then we can implement mutual exclusion by declaring a Boolean variable lock, initialized to false.

If the machine supports the Swap instruction, then mutual exclusion can be provided as follows. A global Boolean variable lock is declared and is initialized to false. In addition, each process also has a local Boolean variable key.

## Semaphores:

The solutions to the critical-section problem presented before are not easy to generalize to more complex problems. To overcome this difficulty, we can use a synchronization tool called a semaphore. A **semaphore** S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations: **wait** and **signal**. These operations were originally termed P (for wait; from the Dutch proberen, to test) and V (for signal; from verhogen, to increment). The classical definition of wait in pseudocode is

```
Wait(S) {
while (S <= 0);
 // no-op
   S --;
    }
```

The classical definitions of signal in pseudocode is

```
Signal(S){
S++;
}
```

Modifications to the integer value of the semaphore in the wait and signal operations must be executed indivisibly. That is, when one process modifies the semaphore value, no other process can simultaneously modify that same semaphore value. In addition, in the case of the wait (S), the testing of the integer value of S (S 5 O), and its possible modification (S--), must also be executed without interruption.

**Usage**

We can use semaphores to deal with the n-process critical-section problem. The n processes share a semaphore, mutex (standing for mutual exclusion), initialized to 1. We can also use semaphores to solve various synchronization problems. For example, consider two concurrently running processes: P1 with a statement S1 and P2 with a statement S2. Suppose that we require that S2 be executed only after S1 has completed. We can implement this scheme readily by letting P1 and P2 share a common semaphore synch, initialized to 0, and by inserting the statements in process P1, and the statements

```
wait (synch) ;
s2;
```

in process P2. Because synch is initialized to 0, P2 will execute S2 only after P1 has invoked signal (synch) , which is after S1.

```
s1;
signal (synch) ;
 do {
        wait (mutex) ;
                critical section
         signal (mutex) ;
                remainder section
} while (1);
```

Mutual-exclusion implementation with semaphores.

**Implementation**

The main disadvantage of the mutual-exclusion solutions and of the semaphore definition given here, is that they all require busy waiting. While a process is in its critical section, any other process that tries to enter its critical section must loop continuously in the entry code. This continual looping is clearly a problem in a real multiprogramming system, where a single CPU is shared among many processes. Busy waiting wastes CPU cycles that some other process might be able to use productively. This type of semaphore is also called a **spinlock** (because the process "spins" while waiting for the lock). Spinlocks are useful in multiprocessor systems. The advantage of a spinlock is that no context switch is required when a process must wait on a lock, and a context switch may take considerable time. Thus, when locks are expected to be held for short times, spinlocks are useful.

To overcome the need for busy waiting, we can modify the definition of the wait and signal semaphore operations. When a process executes the wait operation and finds that the semaphore value is not positive, it must wait. However, rather than busy waiting, the process can block itself. The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state. Then, control is transferred to the CPU scheduler, which selects another process to execute.

A process that is blocked, waiting on a semaphore S, should be restarted when some other process executes a signal operation. The process is restarted by a wakeup operation, which changes the process from the waiting state to the ready state. The process is then placed in the ready queue. (The CPU may or may not be switched from the running process to the newly ready process, depending on the CPU-scheduling algorithm.)

To implement semaphores under this definition, we define a semaphore as a "C" struct:

```
typedef struct {
int value ;
struct process *L;
} semaphore;
```

Each semaphore has an integer value and a list of processes. When a process must wait on a semaphore, it is added to the list of processes. A signal operation removes one process from the list of waiting processes and awakens that process.

The wait semaphore operation can now be defined as:

```
void wait(semaphore S)
```

```
{
      S.value--;
      if (S.value < 0)
      {
      add this process to S.L;
      }
      block() ;
}
```

The signal semaphore operation can now be

defined as void signal(semaphore S)

```
{
      S.value++;
      if (S.value <= 0) {
      remove a process P from S.L;
      }
      wakeup (P1) ;
}
```

The block operation suspends the process that invokes it. The wakeup(P1) operation resumes the execution of a blocked process P. These two operations are provided by the operating system as basic system calls.

**Binary Semaphores**

The semaphore construct described in the previous sections is commonly known as a counting semaphore, since its integer value can range over an unrestricted domain. A binary semaphore is a semaphore with an **integer value that can range only between 0 and 1**. A binary semaphore can be simpler to implement than a counting semaphore, depending on the underlying hardware architecture. We will now show how a counting semaphore can be implemented using binary semaphores. Let S be a counting semaphore.

To implement it in terms of binary semaphores we need the following data structures:

```
binary-semaphore S1, S2;
int C;
```

Initially S1 = 1, S2 = 0, and the value of integer C is set to the initial value of the counting semaphore S. The wait operation on the counting semaphore S can be implemented as follows:

```
wait (S1) ;
C--;
i f (C < 0)
{
  signal(S1) ; wait (S2) ;
}
signal(S1);
```

The signal operation on the counting semaphore S can be implemented as follows:

```
w a i t (S1) ;
C++ ;
i f (C <= 0)
signal (S2) ;
e l s e
signal (S1) ;
```

## Classical Problems of Synchronization:

### 1. The Bounded-Buffer Problem

The bounded-buffer problem is commonly used to illustrate the power of synchronization primitives. We present here a general structure of this scheme, without committing ourselves to any particular implementation. We assume that the pool consists of n buffers, each capable of holding one item. The mutex semaphore provides mutual exclusion for accesses to the buffer pool and is initialized to the value 1. The empty and full semaphores count the number of empty and full buffers, respectively. The semaphore empty is initialized to the value n; the semaphore full is initialized to the value 0.
The code for the producer process is

```
do{
produce an item in nextp
...
wait (empty) ;
wait (mutex) ;
...
add nextp to buffer
. . .
signal(mutex);
signal (full) ;
} while (1);
```

The code for the consumer process is:

```
do{
   wait (full) ;
   wait (mutex) ;
   ….
   remove an item from buffer to nextc
   …..
   signal (mutex) ;
   signal (empty) ;
   ...
   consume the item in nextc
   ...
   } while (1);
```

Note the symmetry between the producer(p) and the consumer(c). We can interpret this code as the producer producing full buffers for the consumer, or as the consumer producing empty buffers for the producer.

### 2. The Readers- Writers Problem

A data object (such as a file or record) is to be shared among several concurrent processes. Some of these processes may want only to read the content of the shared object, whereas others may want to update (that is, to read and write) the shared object. We distinguish between these two types of processes by referring to those processes that are interested in only reading as readers, and to the rest as writers. Obviously, if two readers access the shared data object simultaneously, no adverse effects will result. However, if a writer and some other process (either a reader or a writer) access the shared object simultaneously, chaos may ensue.

To ensure that these difficulties do not arise, we require that the writers have exclusive access to the shared object. This synchronization problem is referred to as the readers-writers problem. Since it was originally stated, it has been used to test nearly every new synchronization primitive. The readers-writers

problem has several variations, all involving priorities. The simplest one, referred to as the first readers-writers problem, requires that no reader will be kept waiting unless a writer has already obtained permission to use the shared object. In other words, no reader should wait for other readers to finish simply because a writer is waiting. The second readers-writers problem requires that, once a writer is ready, that writer performs its write as soon as possible. In other words, if a writer is waiting to access the object, no new readers may start reading.

A solution to either problem may result in starvation. In the first case, writers may starve; in the second case, readers may starve. For this reason, other variants of the problem have been proposed. In this section, we present a solution to the first readers-writers problem.
In the solution to the first readers-writers problem, the reader processes share the following data structures:

```
semaphore mutex, wrt;
 int readcount;
```

The semaphores mutex and wrt are initialized to 1; readcount is initialized to 0. The semaphore wrt is common to both the reader and writer processes. The mutex semaphore is used to ensure mutual exclusion when the variable readcount is updated. The readcount variable keeps track of how many processes are currently reading the object. The semaphore wrt functions as a mutual-exclusion semaphore for the writers. It is also used by the first or last reader that enters or exits the critical section. It is not used by readers who enter or exit while other readers are in their critical sections.

The code for a writer process is

```
do{
wait (wrt) ;
. . .
writing is performed
...
signal(wrt);
}while(1);
```

The code for a reader process is

```
do{
wait (mutex) ;
readcount++;
if (readcount == 1)
wait (wrt) ;
signal (mutex) ;
. . .
reading is performed
...
wait (mutex) ;
readcount--;
if (readcount == 0)
signal(wrt);
signal (mutex) ;
}while(1);
```

Note that, if a writer is in the critical section and n readers are waiting, then one reader is queued on wrt, and n - 1 readers are queued on mutex. Also observe that, when a writer executes signal (wrt), we may resume the execution of either the waiting readers or a single waiting writer.

## The Dining-Philosophers Problem

Consider five philosophers who spend their lives thinking and eating. The philosophers share a common circular table surrounded by five chairs, each belonging to one philosopher. In the center of the table is a bowl of rice, and the table is laid with five single chopsticks. When a philosopher thinks, he does not interact with his colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to his (the chopsticks that are between his and his left and right neighbors). A philosopher may pick up only one chopstick at a time. Obviously, she cannot pick up a chopstick that is already in the hand of a neighbor. When a hungry philosopher has both his chopsticks at the same time, he eats without releasing his chopsticks. When he is finished eating, he puts down both of his chopsticks and starts thinking again.

The structure of philosopher i

```
do {
wait (chopstick[i]) ;
wait (chopstick[(i+1) % 5] ) ;
...
eat
. . .
signal (chopstick [i]) ;
signal(chopstick[(i+1) % 5] ) ;
. . .
think
...
} while (1);
```

The dining-philosophers problem is considered a classic synchronization problem, neither because of its practical importance nor because computer scientists dislike philosophers, but because it is an example of a large class of concurrency-control problems. It is a simple representation of the need to allocate several resources among several processes in a deadlock- and starvation free manner.

One simple solution is to represent each chopstick by a semaphore. A philosopher tries to grab the chopstick by executing a wait operation on that semaphore; he releases his chopsticks by executing the signal operation on the appropriate semaphores. Thus, the shared data are semaphore chopstick ; where all the elements of chopstick are initialized to 1.

Although this solution guarantees that no two neighbors are eating simultaneously,
it nevertheless must be rejected because it has the possibility of creating a deadlock. Suppose that all five philosophers become hungry simultaneously, and each grabs his left chopstick. All the elements of chopstick will now be equal to 0. When each philosopher tries to grab his right chopstick, he will be delayed forever.

Use an asymmetric solution; that is, an odd philosopher picks up first his left chopstick and then his right chopstick, whereas an even philosopher picks up his right chopstick and then his left chopstick. Finally, any satisfactory solution to the dining-philosophers problem must guard against the possibility that one of the philosophers will starve to death.

## Monitors:

Monitors are a synchronization construct that were created to overcome the problems caused by semaphores such as timing errors.

Monitors are abstract data types and contain shared data variables and procedures. The shared data variables cannot be directly accessed by a process and procedures are required to allow a single process to access the shared data variables at a time.

This is demonstrated as follows:

```
monitor monitorname
{
   data variables:
   procedure P1(……)
   {
```

```
  }
Procedure P2(……)
  {

  }
.
.
.
Procedure Pn(…..)
  {

 }
Initialization code(…..)
  {

  }
}
```

Only one process can be active in a monitor at a time. Other processes that need to access the shared variables in a monitor have to line up in a queue and are only provided access when the previous process release the shared variables.

# Module-4

**Deadlocks - System Model, Dead locks Characterization, Methods for Handling Deadlocks Deadlock Prevention, Deadlock Avoidance, Deadlock Detection, and Recovery from Deadlock.**

## Class Note

**Shell Programming: Concept of shell, Types of shell, Editors for shell programming (e.g. vi), basics of Shell programming.**

### LAB Note

**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.

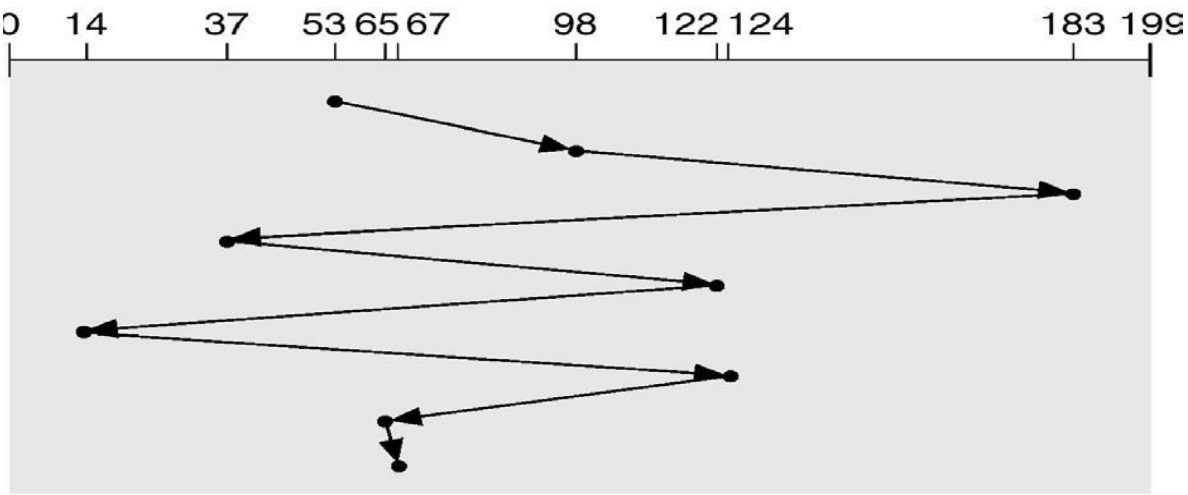
We can improve both the access time and the bandwidth by scheduling the servicing of disk I/O requests in a good order. Several algorithms exist to schedule the servicing of disk I/O requests as follows:

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.



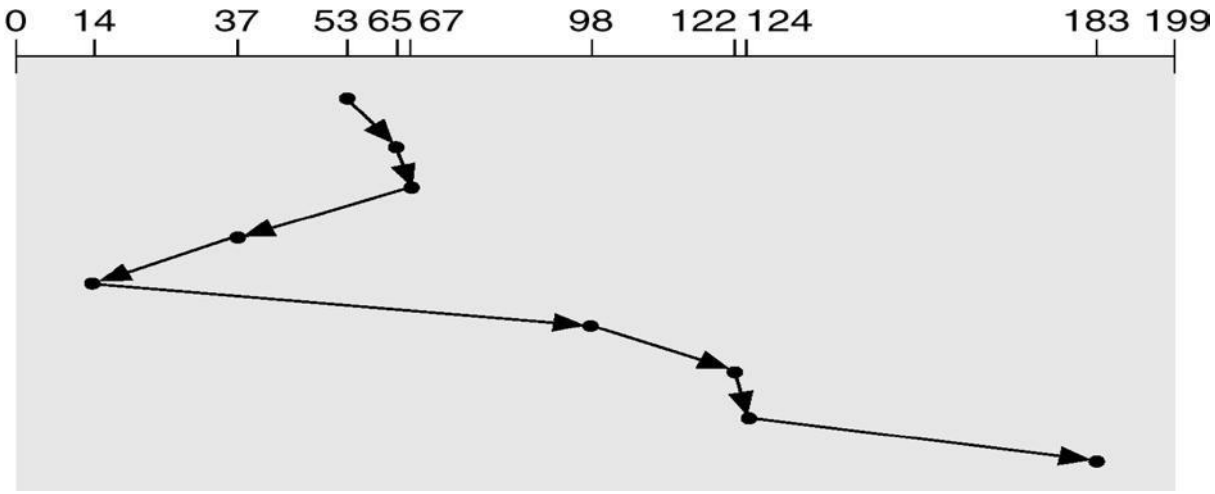

(FCFS disk scheduling)

## 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.

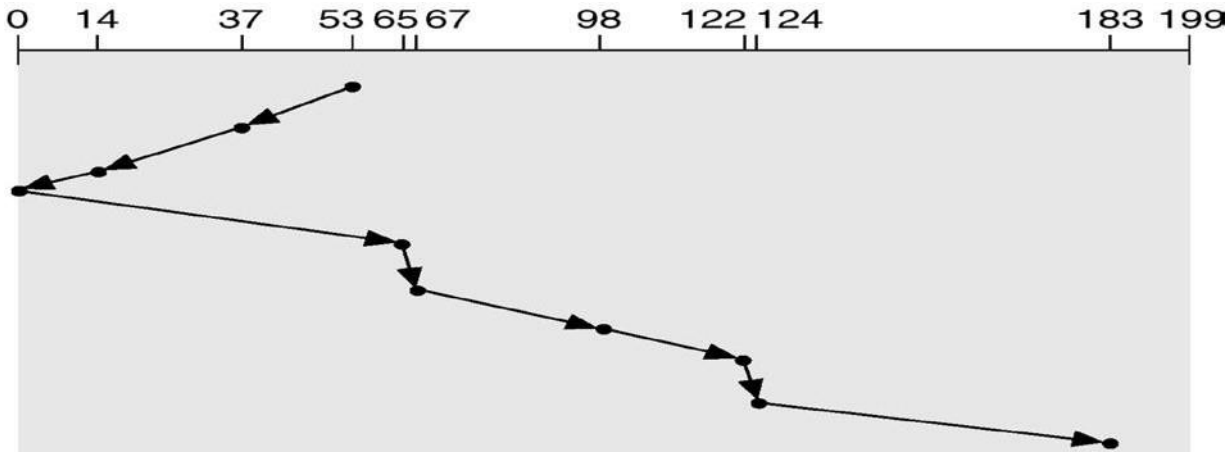



(SSTF disk scheduling)

## 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

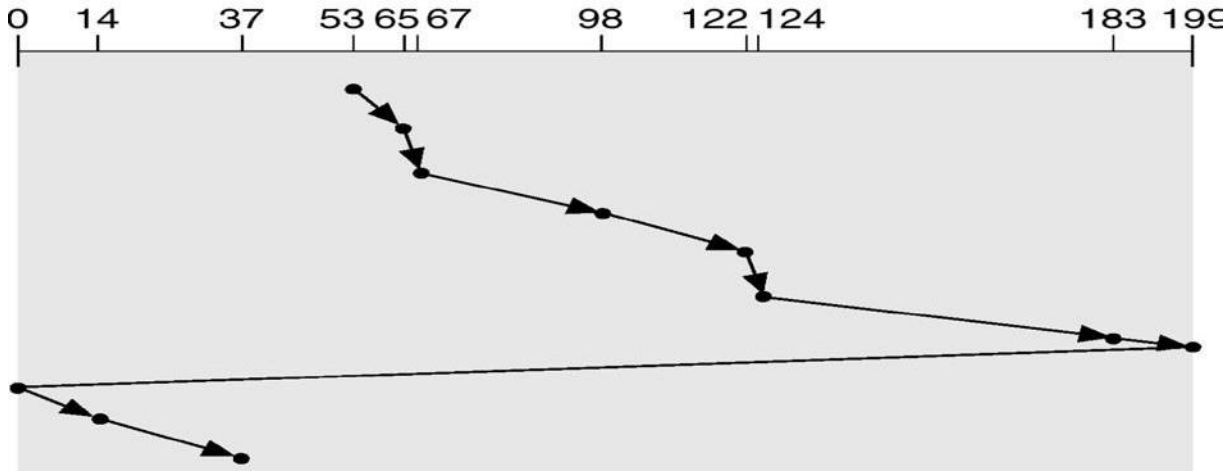
Consider now the Head pointer is in cylinder 53.




(SCAN disk scheduling)

## 4. C-SCAN Scheduling

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

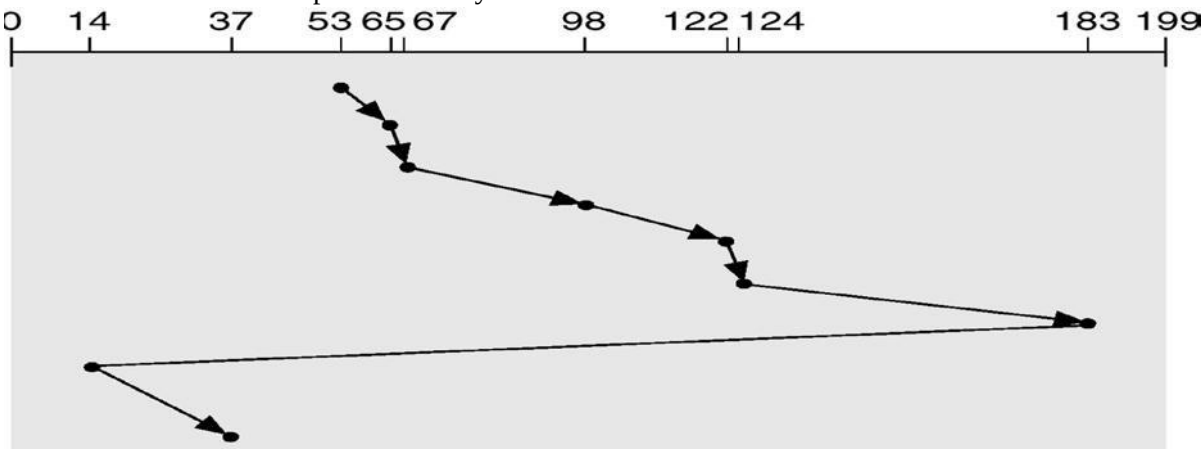



(C-SCAN disk scheduling)

## 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.




(LOOK disk scheduling)

## 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.