BCA I YEAR I SEMESTER

BCA1.5 OPERATING SYSTEM

TABLE OF CONTENTS

UNIT I

Introduction to Operating System:

In the simplest scenario, the operating system is the **first piece of software to run on a computer when it is booted**. Its job is to coordinate the execution of all other software, mainly user applications. It also provides various common services that are needed by users and applications. The operating system is very important part of almost every computer system.

- The operating system has two objectives such as:
  - Firstly, it **controls** the **computer's hardware**.
  - And, finally it provides an interactive **interface between the user and the hardware**.
- It **provides** the **base for application program** to operate.
- An operating system acts as a **resource manager** and allocates resources to specific programs and users as necessary for their task. The commonly required resources are Input/output devices, memory, file storage space, CPU time and so on. Since there may be many, possibly conflicting, requests for resources, the operating system must decide which requests are allocated resources to operate the computer system efficiently.
- An operating system is a **control program**. A control program controls the execution of user programs **to prevent errors and improper use of the computer**. It is especially concerned with the operation and control of I/O devices.

1.1STRUCTURE OF OPERATING SYSTEM

The structure of the operating system consists of 4 layers. There are as follows:

- **Hardware** – CPU, memory, I/O devices and Secondary storage
- **Software** – Operating system program
- **System Programs** – compiler, assembler, linker etc.
- **Application Programs** – programs through which users can interact with computer system, this consist of database system, video games, business programs etc.

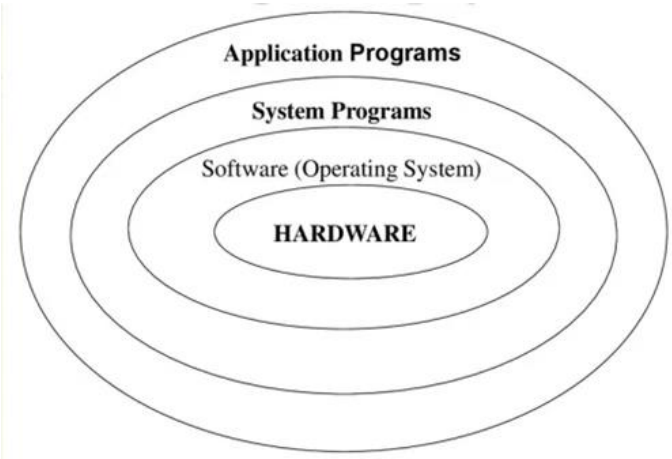


Fig 1.1 Structure of an Operating System

1.2 OPERATING SYSTEM FUNCTIONS

An operating system provides a user-friendly environment for the creation and execution of programs. The operating system provides certain services to programs and to the users of those programs. The operating system functions are as follows:

1. **Program Creation:** The OS provides editors and debuggers to assist the programmer to create programs.

2. **Program Execution**: Users will want to execute programs. There are a number of tasks in each of these programs. And these tasks include instructions and data needs to be loaded in the memory. The I/O devices and files must be initialized and the other resources must be prepared. The program must be able to end its execution either normally or abnormally.

3. **Input/output operations:** A running program may require input and output. This I/O may involve a file or an I/O device. Since a user program cannot execute I/O operations directly, the operating system must provide some means to do so.

4. **Error Detection:** The operating system may detect any kind of error and should be able to take appropriate action for them. Errors may occur in the CPU and memory hardware such as a memory error or power failure and in Input/output devices such as a printer out of paper or in the user program such as an arithmetic overflow or access to illegal memory location.

5. **Resource allocation:** When there are multiple users or multiple jobs running at the same time, resources must be allocated to each of them. Many different types of resources are managed by the operating system. The resources include CPU cycles, main memory, i/o devices, file storage, etc.

6. **Accounting:** In multi-user system, operating system keep track of which user uses how many and which kind of computer resources. It improves the computing services.

7. **Protection:** In multi-user system, when jobs of more than one user executed simultaneously, it should not be possible for one job to interfere with the other. It is also important that a user does not have unauthorized access the files in the network.

1.3 CHARACTERISTICS OF MODERN OS

Huge lines of code and complex features are not the features of a good OS. A good modern OS needs to have the following features:

- **Object Oriented design** – This is the latest trend in designing OS. Here, each facility gets multiple objects created by the OS and each of the objects has a set of operations.
- **Multi-Threading** – An application or program is divided into a number of smaller tasks. These tasks are executed by the process concurrently (simultaneously). These smaller tasks of the program are called threads. A thread is a dispatchable unit of work or a light weight process. The threads have some features of a process but while threads

can share memory space, processes cannot. When the OS executes multiple independent threads of an application at a time, it is called Multi-Threading.

- **Symmetric Multiprocessing –** If a computer is having more than processor, and these processors share memory and I/O resources and these processes share the same job for execution, then this system is called a symmetric multiprocessor system. The operating system designed for such systems are called Symmetric Multiprocessing OS. These systems have a few advantages to uni-processor systems –
    - **Throughput improves –** Throughput – the number of jobs executed by a processor in a time slot.
    - **Reliability –** Even if one processor fails, other processors may take up the tasks and complete them.
    - User, from his end, feels the system responding faster.

- **Distributed Operating System –** Here, the OS runs on a network of computers. The OS, memory files are shared between the different users in the network from the server. The user may feel that he is using a large system with a single OS, and the users need not know where on the network their files are situated.
- **Micro Kernel Architecture –** The minimal OS that performs the essential functions of an OS. System processes perform the other OS functions. These system processes take place in a server. This is a client-server model.

1.4 PROCESS

Process is a program at the time of execution. But each process is more than the program code – it represents the value of program computer (a register in ALU), the process stack which contains temporary data (such as function parameters, return addresses, and local variables), the data in the process register, etc., and, a data section which contains global variables. A process may also include a heap, which is memory that is dynamically allocated during process run time.

Parameter	Process	Program
Definition	Executing part of a program is called a process.	Group of ordered operations to achieve programming goal.
Consists of	Instruction in Programming Language	Instruction Executions – Machine code
Nature	Active - The process is an instance of the program being executed.	Passive - does not do anything until it gets executed.
Resource management	The resource requirement is quite high	The program only needs memory for storage.
Overheads	Processes have considerable overhead.	No significant overhead cost.
Lifespan	Limited - The process has a shorter and very limited lifespan as it gets terminated after the completion of the task.	Unlimited - A program has a longer lifespan as it is stored in the memory until it is not manually deleted.

Creation	New processes require duplication of the parent process.	No such duplication is needed.
Required Process	Process holds resources like CPU, memory address, disk, I/O, etc.	The program is stored on disk in some file and does not require any other resources.
Object type	A process is a dynamic or active entity.	A program is a passive or static entity.
Reside in	Main Memory	Second storage device

1.5 PROCESS MANAGEMENT

In order to accomplish its task, process needs the computer resources. There may exist, more than one process in the system which may require the same resource at the same time. Therefore, the operating system has to manage all the processes and the resources in a convenient and efficient way. Some resources may need to be executed by one process at one time to maintain the consistency otherwise the system can become inconsistent and deadlock may occur.

The operating system is responsible for the following activities in connection with Process Management:

1. Scheduling processes and threads on the CPUs.
2. Creating and deleting both user and system processes.
3. Suspending and resuming processes.
4. Providing mechanisms for process synchronization.
5. Providing mechanisms for process communication.

1.5.1 PROCESS STATES

As a process executes, it changes state. The state of a process is defined in part by the **current activity of** that **process**. The process, from its creation to completion, passes through various states. The minimum number of states is five.

It is important to realize that only one process can be running on any processor at any instant. Many processes may be ready and waiting. A queue is a type of data structure into which the processes are loaded.

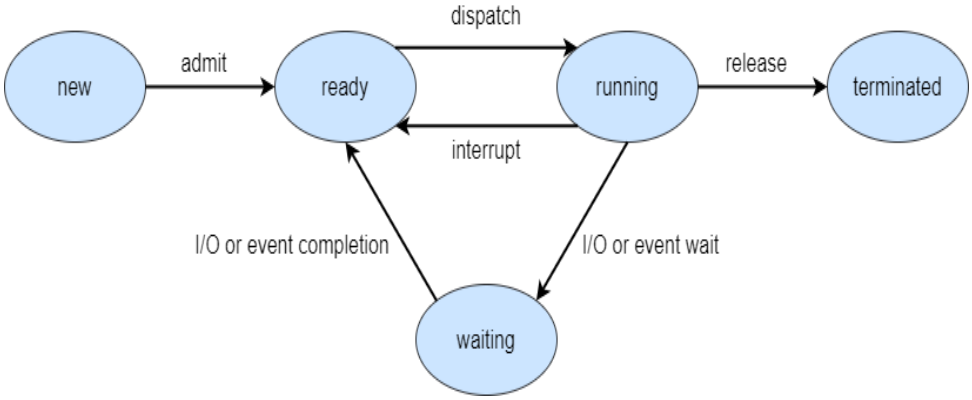


Fig 1.2 Process State Diagram

Each process may be in one of the following states:

- **New**: The process is being created.
- **Running**: Instructions are being executed.
- **Waiting**: The process is waiting for some event to occur (such as an I/O completion or reception of a signal).
- **Ready**: The process is waiting to be assigned to a processor.
- **Terminated**: The process has finished execution.

New ⟶ Ready: The operating system creates a process and prepares the process to be executed. Once prepared, the operating system moved the process into Ready state.

Ready ⟶ Running: The operating system selects one of the job from the ready queue and move the process from ready state to running state.

Running ⟶ Terminated: When the execution of a process has completed, then the operating system terminates that process from running state. Sometimes operating system terminates the process due to some other reasons like memory unavailable, access violation, protection error, I/O failure, data misuse and so on.

Running ⟶ Ready: When the time slot of the processor expired, or if the processor received any interrupt signal, then the operating system shifts running process to ready state.

Running ⟶ Waiting: A process is put in to the waiting state, if the process needs an event to occur, or requires an I/O device or a resource. The operating system does not provide the I/O or event immediately then the process moved to waiting state by the operating system.

Waiting ⟶ Ready: A process in the blocked state is moved to the ready state when the event for which it has been waiting occurs. For example a process is in running state need an I/O device. Then the process moved to the blocked or waiting state, when the I/O device provided by the operating system the process moved to ready state from waiting or blocked state.

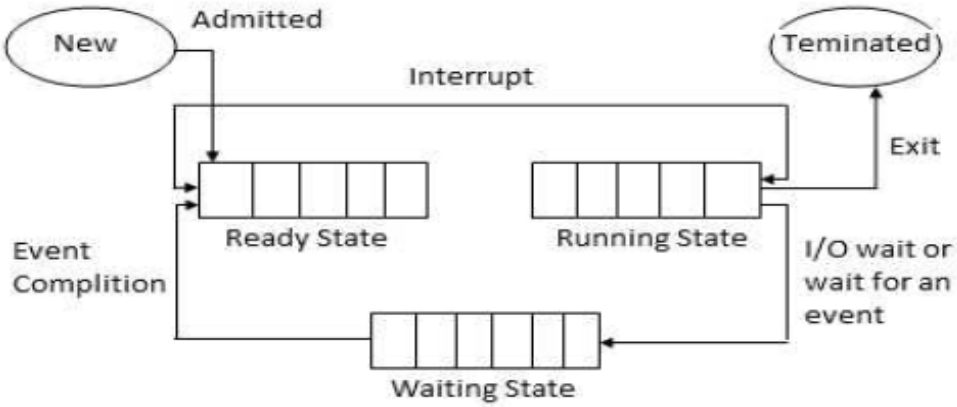


Fig 1.3 Process states with queues

1.5.2 CREATION

One of the main operations on process is to create a new one. The “process-create” call can be used to create a new process. When a user initiates to execute a program, the OS

creates a process to represent the execution of the program. The creation of executable program includes many steps.

The source modules or **source code** (a program written in programming languages) **is translated** in **to object code** or object modules with the help of **translator (compiler)**. The relocatable object modules converted to **absolute programs by linker**. The absolute programs are converted into **executable programs by loaders**. Then the processor executes this program. This **executing program is** called **process**. The process consists of the machine code image of the program in memory and PCB structure.

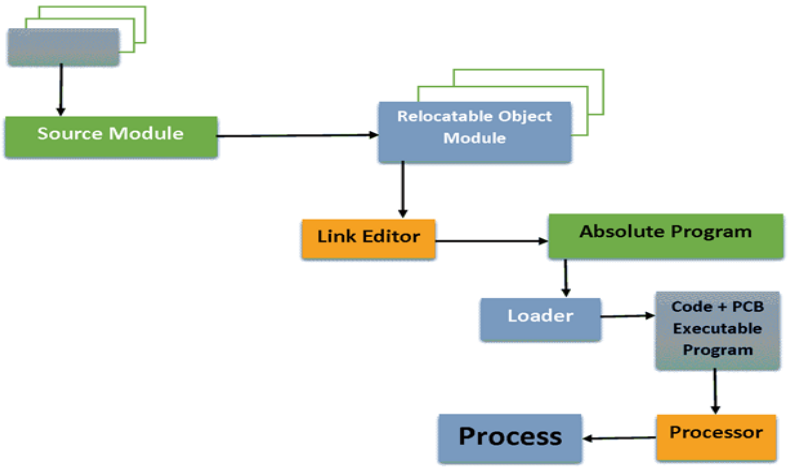


Fig 1.4 Steps to create a process

Process Control Block

The process consists of the machine code image of the program in memory and PCB structure. **Each process is represented in the operating system by a process control block (PCB)—also called a task control block.** The contents of PCB vary from process to process.

A PCB is a data block that holds several information of a specific process, including these:

- **Process state**: The state may be new, ready, running, waiting, halted, and, so on.
- **Program counter**: The counter indicates the address of the next instruction to be executed for this process.
- **CPU registers**: The registers vary in number and type, depending on the computer architecture. They include accumulators, index registers, stack pointers, and general-purpose registers, plus any condition-code information. Along with the program counter, this state information must be saved when an interrupt occurs.
- **CPU-scheduling information**: This information includes a process priority, pointers to scheduling queues, and any other scheduling parameters.
- **Memory-management information**: This information may include such information as the value of the base and limit 30 registers, the page tables, or the segment tables, depending on the memory system used by the operating system
- **Accounting information**: This information includes the amount of CPU and real time used, time limits, account numbers, job or process numbers, and so on.

- **I/O status information**: This information includes the list of I/O devices allocated to the process, a list of open files, and so on. The contents of PCB vary from process to process.

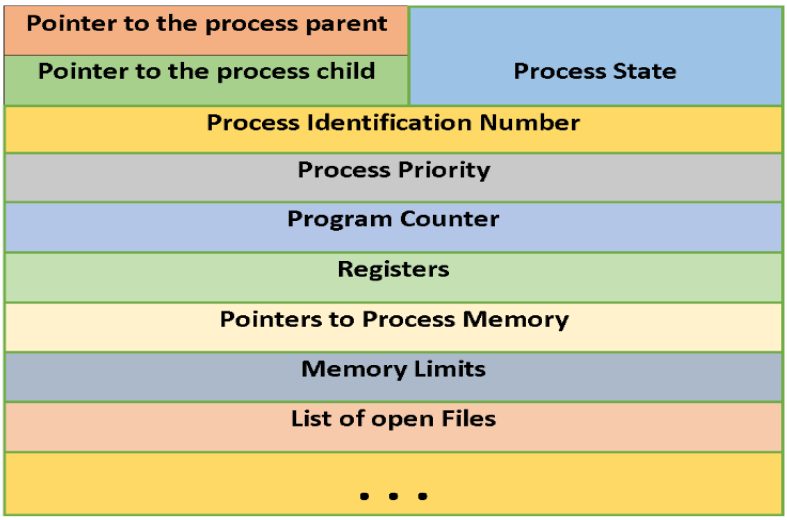


Fig 1.5 Process Control Block

The PCB is a central store of information that allows the operating system to locate all the key information about the process. The operating system uses this information and performs the operations on the process. The operations include suspend a process, resume a process, change the process priority, name the process, dispatch a process, etc.

Spawning

Process spawning is a technique in which OS creates a child process by the request of another process. The process that creates a new process is called the parent process. The child process itself can spawn another process to become a parent. This leads to a tree of processes, yielding a "Hierarchy Process structure".

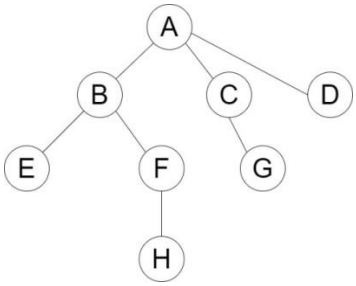


Fig 1.6 A Hierarchical Process Structure

1.5.3 TERMINATION

The dual of creating a process is terminating it. A process can terminate itself by returning from its main function, or by calling the exit system call. Generally the process terminates when execution finished. Some other causes are:

- **Time slot expired**: When the process execution does not complete within the time slot, then the process is terminated from running state. The CPU picks next job in ready queue to execute.

- **Memory violation**: If the process needs more memory than available memory, then the process terminated from running state.
- **I/O Failure**: A process needs an I/O operation at the time of execution, but the I/O device is not available at that time. Then the process is moved to waiting state. The operating system does not provide the I/O device, even though the process resides in the waiting state, the process is terminated.
- **Invalid instruction**: If the process has invalid instructions and the CPU fails to execute those instructions, then the Process terminated.
- **Parent Termination**: When the parent process is terminated, the child process is also terminated automatically
- **Parent Request**: The parent process also has the authority to terminate any of its child processes. The parent can request the termination of a child process.

1.5.4 OPERATIONS ON PROCESS

Systems that manage processes must be to perform certain operations on them. These include:

- **Create** a process – New to Ready
- **Terminate** a process – Running to Terminated
- **Resume** a process (restart the process) – Suspended to Ready
- **Change** the process **priority** – Change process priority in PCB
- **Block** a process – Running to Waiting
- **Wakeup** a process – Waiting to Ready
- **Dispatch** a process – Ready to Running
- **Enable** a process **to communicate** with other processes.

1.5.5 CONCURRENT PROCESS

Concurrent processing is a computing model in which multiple processors execute instructions simultaneously for better performance. Tasks are broken down into subtasks that are then assigned to separate processors to perform simultaneously, instead of sequentially (where one process starts only after the first is completed), as they would have to be carried out by a single processor. Concurrent processes can have their execution overlapping with respect to time. Process execution, although concurrent, is usually not independent. Processes may affect each other's behaviour through shared data, shared resources, communication, and synchronization.

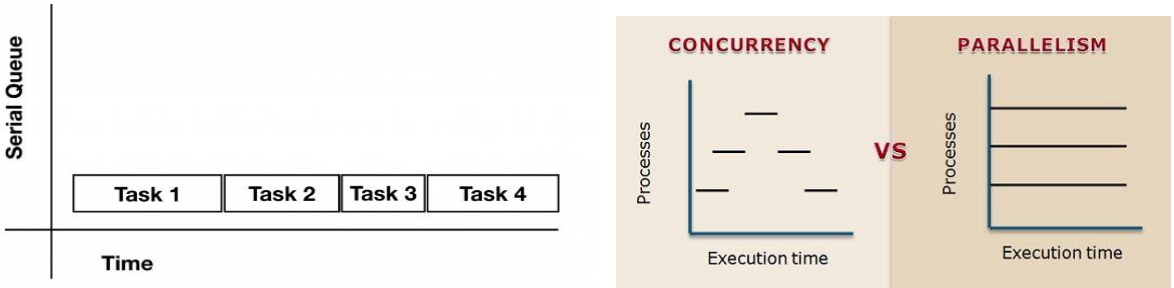


Fig 1.7 Tasks/ Processes performed in different styles of execution.

In figure (a) process1 starts its execution at time t0, process2 starts its execution only after process1 finished its execution, and process3 starts its execution only when process2 finished its execution. These processes are said to be serial processes. In figure (b), the execution time of process1, process2, process3 are overlapped, so these are said to be concurrent processing.

1.5.6 PROCESS THREADS

In computer science, a thread of execution is the smallest sequence of programmed instructions that can be managed independently by an operating system scheduler. A thread is a light-weight process. The implementation of threads and processes differs from one operating system to another, but in most cases, a thread is contained inside a process. Multiple threads can exist within the same process and share resources such as memory, while different processes do not share these resources.

A process is divided into number of light weight processes - threads. The thread has a program counter that keeps the track of which instruction to execute next, it has registers, which holds its current working variables. It has a stack, which contains the execution history.

Number of threads can share memory space, open files and other resources. Same as number of processes can share physical memory, disk, printers and, so on. Thread has some of the properties of process and it operates in many respects in the same manner as process. Threads can be in one of several states: Ready, Blocked, Running or Terminated. Threads share the CPU, and only one thread can be active at a time. Threads can create child threads but are not independent of one another. All threads can access every address in the task. A thread can read or write over any other threads stacks. The thread of program which may be executed concurrently with other thread, this capability is called multi-threading.

Life cycle of a thread:

1. **Born** state: A thread has just been created.
2. **Ready** state: The thread is waiting for processor – CPU
3. **Running** state: The thread is being executed by processor.
4. **Blocked** state: The thread is waiting for an event to occur or for a resource
5. **Sleep** state: Sleeping thread will be ready only after designated sleeping time is over.
6. **Dead** state: The thread execution is completed.

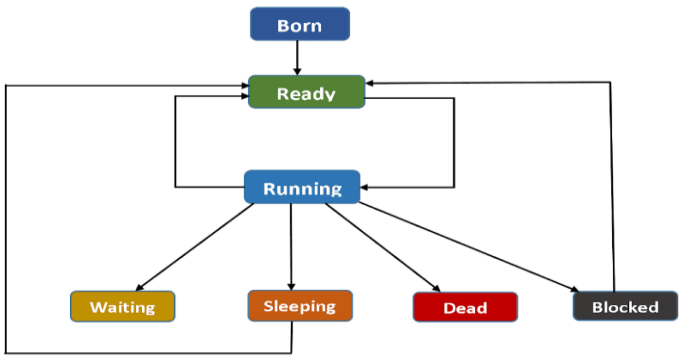


Fig 1.8 Life Cycle of a Thread

Example of thread:

Programmer wish to type the text in word processor, then the programmer open a file in word processor, and typing the text, it is one thread, then the text is automatically formatting, it is another thread. The text is automatically specifies the spelling mistakes, is again another thread. And finally the file is automatically saved in the disk, which is again a thread. Thus typing, formatting, spell checking, saving file all these are threads.

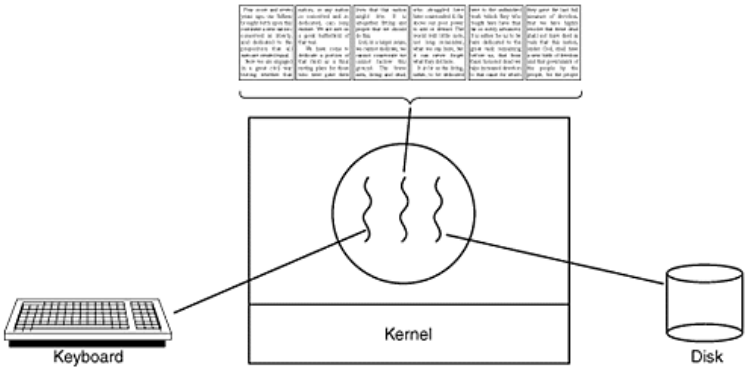


Fig 1.9 Word processor with 3 threads

1.5.7 MULTITHREADING

A process is divided into number of smaller tasks, each task is called thread. When a number of threads within a process execute at a time is called Multithreading. Based on the functionality threads are divided into four categories:

1) **One to one** – (**One process one thread**) – Here, one process maintains only one thread, so it is called as single threaded approach. MS-DOS operating system supports this type of approach.
2) **One to Many** – (**One process, multiple threads**) – A process is divided into number of threads. The best example of this is JAVA run time environment.
3) **Many to one** – (**Multiple processes, one thread per process**) – An operating system support multiple user processes but only support one thread per process. Best example is UNIX.
4) **Many to many** – (**Multiple processes, multiple threads per process**) – In this approach each process is divided into number of threads. Examples are Windows 2000, Solaris LINUX.

Benefits of a Thread:

- Threads minimize the context switching time within the same process.
- Use of threads provides concurrency within a process.
- Efficient communication between threads.
- It takes less to create new threads within the same process.
- Threads allow utilization of multiprocessor architectures to a greater scale and efficiency.
- Threads take less time to terminate than process.

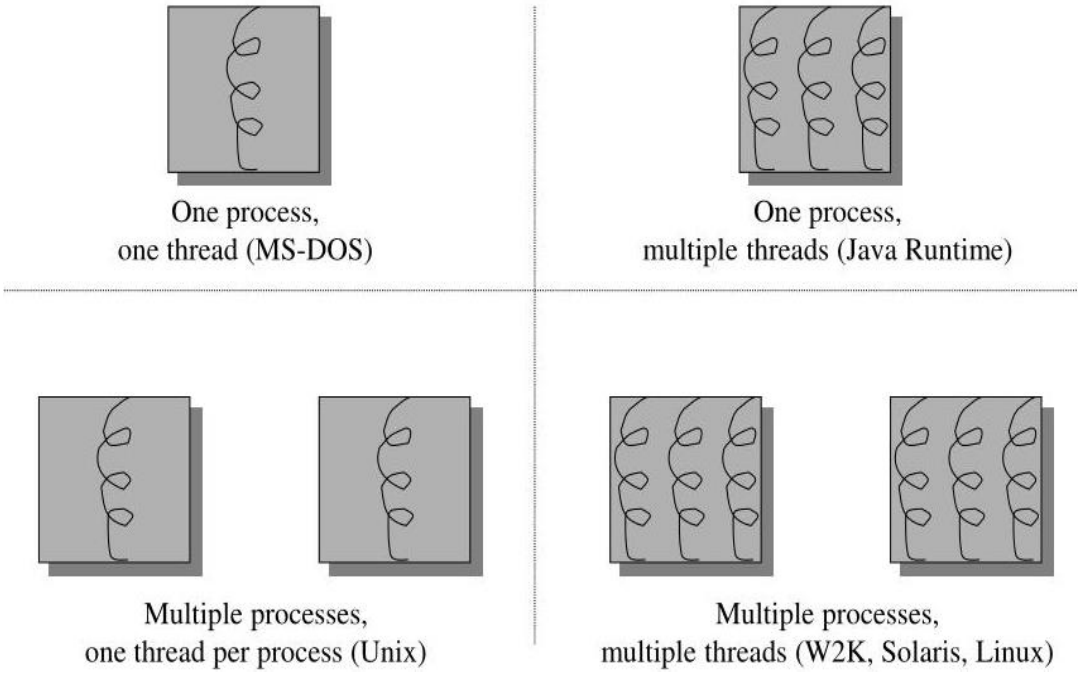


Fig 1.10 The 4 categories of based on functionality

Thread Functionality:

- **Thread states**
- **Thread synchronization** – All threads within a process share memory and files. One thread may make changes in a value of a file/ register and this might affect other threads. Hence, this is a critical issue to be sure that threads that access shared variables or resources are coordinated.

Process vs. Thread

Comparison Basis	**Process**	**Thread**
Definition	A process is a program under execution i.e an active program.	A thread is a lightweight process that can be managed independently by a scheduler.
Context switching time	Processes require more time for context switching as they are more heavy.	Threads require less time for context switching as they are lighter than processes.
Memory Sharing	Processes are totally independent and don't share memory.	A thread may share some memory with its peer threads.
Communication	Communication between processes requires more time than between threads.	Communication between threads requires less time than between processes .

Blocked	If a process gets blocked, remaining processes can continue execution.	If a user level thread gets blocked, all of its peer threads also get blocked.
Resource Consumption	Processes require more resources than threads.	Threads generally need less resources than processes.
Dependency	Individual processes are independent of each other.	Threads are parts of a process and so are dependent.
Data and Code sharing	Processes have independent data and code segments.	A thread shares the data segment, code segment, files etc. with its peer threads.
Treatment by OS	All the different processes are treated separately by the operating system.	All user level peer threads are treated as a single task by the operating system.
Time for creation	Processes require more time for creation.	Threads require less time for creation.
Time for termination	Processes require more time for termination.	Threads require less time for termination.

Implementation of Threads – User level and Kernel level Thread

User level threads:

This type of threads loaded entirely in user space, the kernel knows nothing about them. When threads are managed in user space, each process has its own private thread table. The thread table contains information like program counter, registers, state etc. When the thread is moved to ready state or blocked state, the information needed to restart it is stored in the thread table.

Advantages:

- User level threads allow each process to have its own scheduling algorithm.
- The entire process is loaded in user space, so the process does not switch to the kernel mode to do thread management. This saves the overhead of two mode switches.
- User level threads can run on any operating system.

Disadvantages:

- When user level thread executes a system call, not only the participants tread blocked, all the threads within the process are blocked.

- In user level threads, only a single thread with in a process can execute at a time, so multi-processing is cannot implemented.

Kernel level threads:

In kernel level threads, the kernel does total work of thread management. There is no thread table in each process. The kernel has a thread table that keeps track of all the threads in the system. When a thread wants to create a new thread or destroy any existing thread it makes call to the kernel, kernel then takes the action.

The kernel thread table holds each thread register, state and other information. The information is the same as with user level threads, but it is now in the kernel instead of the user space.

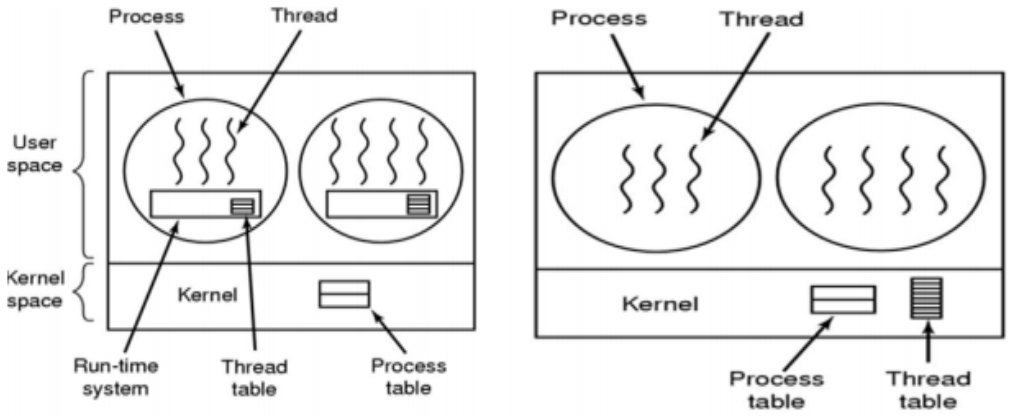


Fig 1.11 User level Thread and Kernel level Threads

Advantages:

- The kernel can simultaneously schedule a multiple threads from the same process on multiple processors.
- If a thread in a process is blocked, then the kernel can schedule another thread of the same process.
- Kernels routines can be multithreaded.
- Scheduling by kernel is done on a thread bus.

Disadvantages:

- It requires more cost for creating and destroying threads in kernel.
- The transfer of control from one thread to another within the same process requires a mode switch to the kernel.

Combines Approaches:

If both the approaches – user and kernel level, then we get the advantages of both the systems. Some OS provides this type of facility e.g. Solaris. In the combined system, the thread creation is done completely in the user space.

1.5.8 MICROKERNELS

A micro kernel is a software program, loaded in main memory, to perform some of the functions of the operating system. So, the size of the OS can be reduced. Micro kernels

provide process scheduling, memory management, and communication services. Initially micro kernel was used in MACH OS. Windows 2000 also uses this approach.

Micro kernel Architecture

In the Micro kernel Architecture, the essential and main operating system functions should be executed in the kernel architecture. Remaining functions are executed in user mode. In the usual OS, all the functions are loaded into the Kernel mode. A micro kernel is a minimal operating system that performs only the essential functions of an operating system. The other functions are performed by the system processes.

Benefits of micro kernel organisation:

- Microkernel architecture is small and isolated therefore it can function better.
- Microkernels are secure because only those components are included that disrupt the functionality of the system otherwise.
- The expansion of the system is more accessible, so it can be added to the system application without disturbing the Kernel.
- Microkernels are modular, and the different modules can be replaced, reloaded, modified without even touching the Kernel.
- Fewer system crashes when compared with monolithic systems.
- Microkernel interface helps you to enforce a more modular system structure.
- Without recompiling, add new features
- Server malfunction is also isolated as any other user program's malfunction.
- Microkernel system is flexible, so different strategies and APIs, implemented by different servers, which can coexist in the system.
- Increased security and stability will result in a decreased amount of code which runs on kernel mode

Disadvantage of Microkernel:

- Providing services in a microkernel system are expensive compared to the normal monolithic system.
- Context switch or a function call needed when the drivers are implemented as procedures or processes, respectively.
- The performance of a microkernel system can be indifferent and may lead to some problems.

1.6 CPU SCHEDULING

The objective of multiprogramming is to have some process running at all times, to maximize CPU utilization. The objective of time sharing is to switch the CPU among processes so frequently that users can interact with each program while it is running. To meet these objectives, the process scheduler selects an available process (possibly from a set of several available processes) for program execution on the CPU. For a single-processor system, there will never be more than one running process. If there are more processes, the rest will have to wait until the CPU is free and can be rescheduled.

Scheduling queues:

As processes enter the system, they are put into a job queue, which consists of all processes in the system. The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the ready queue. This queue is generally stored as a linked list. A ready-queue header contains pointers to the first and final PCBs in the list. Each PCB includes a pointer field that points to the next PCB in the ready queue.

The system also includes other queues. When a process is allocated the CPU, it executes for a while and eventually quits, is interrupted, or waits for the occurrence of a particular event, such as the completion of an I/O request. Suppose the process makes an I/O request to a shared device, such as a disk. Since there are many processes in the system, the disk may be busy with the I/O request of some other process. The process therefore may have to wait for the disk. The list of processes waiting for a particular I/O device is called a device queue. Each device has its own device queue.

1.6.1 SCHEDULERS

There are three main schedulers – Long term scheduler, Short term scheduler and, Medium term scheduler

Long term Scheduler:

The function of long term scheduler is, selects processes from job pool and loads them into main memory (ready queue) for execution, so the long term scheduler is called as job scheduler.

For example, assume that a computer lab consisting of 20 dummy nodes connected to the server using local area network, out of 20, ten users want to execute their jobs, then these ten jobs are loaded into spool disk. The long term scheduler selects the jobs from the spool disk and loaded them into main memory in ready queue.

A ready queue is a data structure in which the data is inserted in the front and deleted from the rear.

Short term Scheduler:

The short-term scheduler, or CPU scheduler, selects a job from ready queue (processes that are ready to execute) and allocates the CPU to that process with the help of dispatcher. The method of selecting a process from the ready queue (by short term scheduler)is depending on CPU scheduling algorithm.

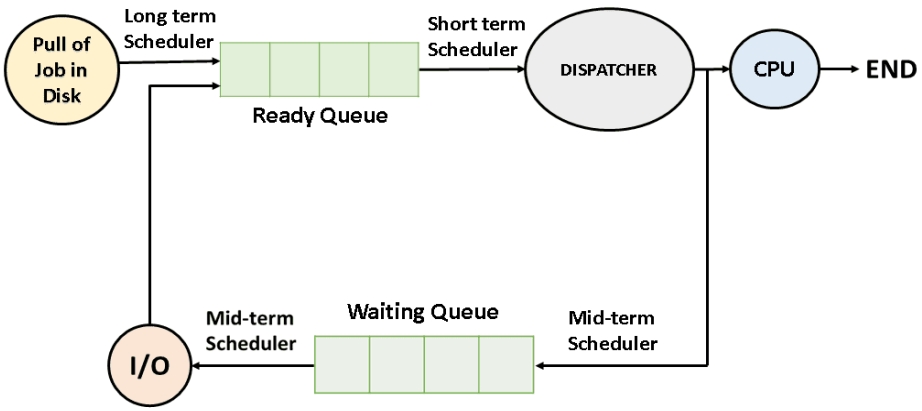


Fig 1.12 Queuing Diagram for the CPU Scheduling

Dispatcher:

Dispatcher is a module, which connects the CPU to the process selected by the short term scheduler. The main function of dispatcher is switching the CPU from one process to another process. Another function of dispatcher is jumping to the proper location in the user program, and ready to start execution. The dispatcher should be fast, because it is invoked during each and every process switch. The time taken by dispatcher to stop one process and start another process is known as _dispatch latency‘. The degree of multiprogramming is depending on the dispatch latency. If the dispatch latency is increasing, then the degree of multiprogramming is decreases.

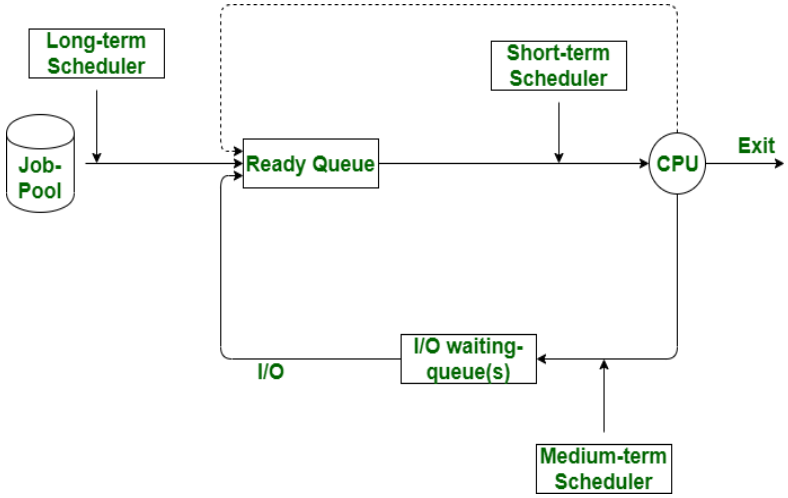


Fig 1.13 Schedulers and process state transitions

Medium term Scheduler

If the process request is an I/O in the middle of the execution, then the process is removed from the main memory and loaded in to waiting queue. When the I/O operation completed, then the job moved from waiting queue to ready queue. These two operations are performed by medium term scheduler.

The short-term scheduler must select a new process for the CPU frequently. A process may execute for only a few milliseconds before waiting for an I/O request. Often, the short term scheduler executes at least once every 100 milliseconds. Because of the short time between executions, the short-term scheduler must be fast. The long-term scheduler executes much less frequently; minutes may separate the creation of one new process and the next. The long-term scheduler controls the degree of multiprogramming (the number of processes in memory). If the degree of multi-programming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system. Thus, the long-term scheduler may need to be invoked only when a process leaves the system.

1.6.2 SCHEDULING METHODOLOGY

Different CPU scheduling algorithms have different properties, and the choice of a particular algorithm may favour one class of processes over another. In choosing which algorithm to use in a particular situation, we must consider the properties of the various algorithms. Many criteria have been suggested for comparing CPU scheduling algorithms.

These criteria are used for comparison between the algorithms to choose the best one. The criteria include the following:

CPU utilization – We want to keep the CPU as busy as possible. The CPU utilization may range from 0 to 100 percent. In a real system, it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily used system).

Throughput – If the CPU is busy executing processes, then work is being done. One measure of work is the number of processes that are completed per time unit, called throughput. For long processes, this rate may be one process per hour; for short transactions, it may be 10 processes per second.

Turnaround time – From the point of view of a particular process, the Important criterion is how long it takes to execute that process. The interval from the time of submission of a process to the time of completion of that process is the turnaround time. Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU, and doing I/O.

Waiting time –The CPU scheduling algorithm does not affect the amount of time during which a process executes or does I/O; it affects only the amount of time that a process spends waiting in the ready queue. Waiting time is the sum of the periods spent waiting in the ready queue.

Response time – Response time is the time from the submission of a request until the first response is produced. It is the time it takes to start responding, not the time it takes to output the response. The turnaround time is generally limited by the speed of the output device. It is desirable to maximize CPU utilization and throughput and to minimize Turnaround time, waiting time, and response time.

1.6.3 CPU SCHEDULING ALGORITHM

1.6.3.1 FIRST COME FIRST SERVE (FCFS)

The simplest CPU-scheduling algorithm is the first come, first served (FCFS) schedule algorithm. With this scheme, the process that requests the CPU first is allocated the CPU first. The implementation of the FCFS policy is easily managed with a FIFO queue. When a process enters the ready queue, its PCB is linked onto the tail of the queue. When the CPU is free, it is allocated to the process at the head of the queue. The running process is then removed from the queue. The code for FCFS scheduling is simple to write and understand. On the negative side, the average waiting time under the FCFS policy is often quite long.

Key features:

- The process which arrives first in the ready queue is firstly assigned the CPU.
- In case of a tie, process with smaller process id is executed first.
- It is always non-pre-emptive in nature.

Advantages-

- It is simple and easy to understand.
- It can be easily implemented using queue data structure.
- It does not lead to starvation.

Disadvantages-

- It does not consider the priority or burst time of the processes.
- It suffers from **convoy effect**.

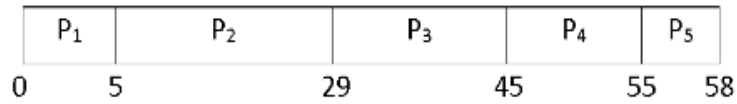
Note: In convoy effect,

- Consider processes with higher burst time arrived before the processes with smaller burst time.
- Then, smaller processes have to wait for a long time for longer processes to release the CPU.

Problem:

Consider the following set of processes that arrive at time 0, with the length of the CPU burst given in milliseconds:

Process	Burst Time(ms)
P_1	5
P_2	24
P_3	16
P_4	10
P_5	3



Consider the above set of processes that arrive at time zero. The length of the CPU burst time given in millisecond. Now we calculate the average waiting time, average turnaround time and throughput.

Average Waiting Time and Turnaround Time

Average Waiting Time

Waiting Time = Starting Time - Arrival Time

Waiting time of
P1 = 0
P2 = 5 - 0 = 5 ms
P3 = 29 - 0 = 29 ms
P4 = 45 - 0 = 45 ms
P5 = 55 - 0 = 55 ms

Average Waiting Time = Waiting Time of all Processes / Total Number of Process

Therefore, average waiting time = (0 + 5 + 29 + 45 + 55) / 5 = 26.8 ms

Average Turnaround Time

Turnaround Time = Waiting time in the ready queue + executing time + waiting time in waiting-queue for I/O

Turnaround time of
P1 = 0 + 5 + 0 = 5ms
P2 = 5 + 24 + 0 = 29ms
P3 = 29 + 16 + 0 = 45ms
P4 = 45 + 10 + 0 = 55ms
P5 = 55 + 3 + 0 = 58ms

Total Turnaround Time = (5 + 29 + 45 + 55 + 58)ms = 192ms

Average Turnaround Time = (Total Turnaround Time / Total Number of Process)

= (192 / 5)ms = 38.4ms

Average Response Time

Response Time = First Response - Arrival Time

Response time of
P1 = 0
P2 = 5 – 0 = 5 ms
P3 = 29 – 0 = 29 ms
P4 = 45 – 0 = 45ms
P5 = 55 – 0 = 55ms

Total Turnaround Time = (5 + 29 + 45 + 55) ms = 134ms

Average Turnaround Time = (Total Turnaround Time / Total Number of Process)

= (134 / 5)ms = 26.8ms

Throughput

Here, we have a total of five processes. Process P1, P2, P3, P4, and P5 takes 5ms, 24ms, 16ms, 10ms, and 3ms to execute respectively.

Throughput = (5 + 24 +16 + 10 +3) / 5 = 11.6ms

It means one process executes in every 11.6 ms.

1.6.3.2 SHORTEST JOB FIRST (SJF)

In this approach the CPU is allocated to the process having the shortest CPU burst time. When the CPU is available, it is assigned to the process that has the smallest next CPU burst. If the next CPU bursts of two processes are the same, FCFS scheduling is used to break the tie.

The SJ F scheduling algorithm is provably optimal, in that it gives the Minimum average waiting time for a given set of processes. Moving a short Process before a long one decreases, the waiting time of the short process more than it increases the waiting time of the long process. Consequently, the average waiting time decreases. The real difficulty with the SJF algorithm is in knowing the length of the next CPU request. There is no way to know the length of the next CPU burst.

Key Features:

- Out of all the available processes, CPU is assigned to the process having smallest burst time.
- In case of a tie, it is broken by FCFS Scheduling.
- SJF Scheduling can be used in both pre-emptive and non-pre-emptive mode.

Advantages-

- SRTF is optimal and guarantees the minimum average waiting time.

- It provides a standard for other algorithms since no other algorithm performs better than it.

Fig 1.13 Types of SJF algorithm modes of execution

Disadvantages-

- It cannot be implemented practically since burst time of the processes cannot be known in advance.
- It leads to starvation for processes with larger burst time.
- Priorities cannot be set for the processes.
- Processes with larger burst time have poor response time

Pre-emptive SJF scheduling algorithm (shortest remaining-time-first):

The SJF algorithm can be either pre-emptive or non-pre-emptive. The choice arises when a new process arrives at the ready queue while a previous process is still executing. The next CPU burst of the newly arrived process may be shorter than what is left of the currently executing process. A pre-emptive SJF algorithm will pre-empt the currently executing process, where as a non-pre-emptive SJF algorithm will allow the currently running process to finish its CPU burst.

1.6.3.3 ROUND ROBIN (RR)

The round-robin (RR) scheduling algorithm is designed especially for time-sharing systems. It is similar to FCFS scheduling, but pre-emption is added to switch between processes. A small unit of time, called a time quantum or time slice, is defined. A time quantum is generally from 10 to 100 milliseconds. The ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.

To implement RR scheduling, we keep the ready queue as a FIFO queue of processes. New processes are added to the tail of the ready queue. The CPU scheduler picks the first process from the ready queue, sets a timer to interrupt after 1 time quantum, and dispatches the process. One of two things will then happen. The process may have a CPU burst of less than 1 time quantum. In this case, the process itself will release the CPU voluntarily. The scheduler will then proceed to the next process in the ready queue.

Otherwise, if the CPU burst of the currently running process is longer than 1 time quantum, the timer will go off and will cause an interrupt to the operating system. A context switch will be executed, and the process will be put at the tail of the ready queue. The CPU scheduler will then select the next process in the ready queue. The average waiting time under the RR policy is often long. CPU is assigned to the process on the basis of FCFS for a fixed amount of time.

Key Features:

- This fixed amount of time is called as time quantum or time slice.
- After the time quantum expires, the running process is pre-empted and sent to the ready queue.
- Then, the processor is assigned to the next arrived process.
- It is always pre-emptive in nature.
- Round Robin Scheduling is FCFS Scheduling with pre-emptive mode.

Advantages-

- It gives the best performance in terms of average response time.
- It is best suited for time sharing system, client server architecture and interactive system.

Disadvantages-

- It leads to starvation for processes with larger burst time as they have to repeat the cycle many times.
- Its performance heavily depends on time quantum.
- Priorities cannot be set for the processes.

Important Notes-

1. With decreasing value of time quantum,
   Number of context switch increases
   Response time decreases
   Chances of starvation decreases

Thus, smaller value of time quantum is better in terms of response time.

2. With increasing value of time quantum,
   Number of context switch decreases
   Response time increases
   Chances of starvation increases

Thus, higher value of time quantum is better in terms of number of context switch.

3. With increasing value of time quantum, Round Robin Scheduling tends to become FCFS Scheduling.
4. When time quantum tends to infinity, Round Robin Scheduling becomes FCFS Scheduling.
5. The performance of Round Robin scheduling heavily depends on the value of time quantum.
6. The value of time quantum should be such that it is neither too big nor too small.

1.6.3.4 PRIORITY SCHEDULING

The SJF algorithm is a special case of the general **priority scheduling algorithm.** A priority is associated with each process, and the CPU is allocated to the process with the highest priority. Equal-priority processes are scheduled in FCFS order. An SJF algorithm is simply a priority algorithm where the priority (p) is the inverse of the (predicted) next CPU

burst. The larger the CPU burst, the lower the priority, and vice versa. Note that we discuss scheduling in terms of *high* priority and *low* priority. Priorities are generally indicated by some fixed range of numbers, such as 0 to 7 or 0 to 4,095. However, there is no general agreement on whether 0 is the highest or lowest priority. Some systems use low numbers to represent low priority; others use low numbers for high priority. This difference can lead to confusion. In this text, we assume that low numbers represent high priority.

Priority scheduling can be either pre-emptive or non-pre-emptive. When a process arrives at the ready queue, its priority is compared with the priority if the currently running process. A pre-emptive priority scheduling algorithm will pre-empt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process. A non-pre-emptive priority scheduling algorithm will simply put the new process at the head of the ready queue. A major problem with priority scheduling algorithms is indefinite blocking, or starvation. A process that is ready to run but waiting for the CPU can be considered blocked. A priority scheduling algorithm can leave some low-priority processes waiting indefinitely. In a heavily loaded computer system, a steady stream of higher-priority processes can prevent a low-priority process from ever getting the CPU. A solution to the problem of indefinite blockage of low priority processes is aging. Aging is a technique of gradually increasing the priority of processes that wait in the system for a long time. For example, if priorities range from 127 (low) to 0 (high), we could increase the priority of a waiting process by 1 every 15 minutes. Eventually, even a process with an initial priority of 127 would have the highest priority in the system and would be executed.

Key Features:

- Out of all the available processes, CPU is assigned to the process having the highest priority.
- In case of a tie, it is broken by **FCFS Scheduling**.
- Priority Scheduling can be used in both pre-emptive and non-pre-emptive mode.

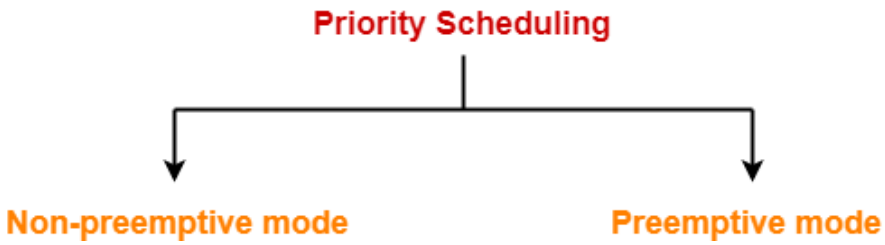


Fig 1.14 Types of Priority Scheduling algorithm modes of execution

Advantages-

- It considers the priority of the processes and allows the important processes to run first.
- Priority scheduling in pre-emptive mode is best suited for real time operating system.

Disadvantages-

- Processes with lesser priority may starve for CPU.
- There is no idea of response time and waiting time.

Important Note:

- The waiting time for the process having the highest priority will always be zero in pre-emptive mode.
- The waiting time for the process having the highest priority may not be zero in non-pre-emptive mode.

Priority scheduling in pre-emptive and non-pre-emptive mode behaves exactly same under following conditions-

- The arrival time of all the processes is same
- All the processes become available

1.7 Questions

QUESTIONS

1. What is an operating system? Explain its structure.
2. What are the main functions of an OS?
3. Write short notes on Multiprogramming and Time sharing systems.
4. Explain the different characteristics of Modern OS.
5. What are the components of an OS
6. What is an operating system?
7. Define Kernel.
8. What is the need for an OS?
9. Short notes on microkernel.
10. There are _____ components in a computer system.
11. Mathematical and logical functions are carried out by the _____.
12. _____ is the backbone of a computer system.
13. Barcode scanners are examples of _____ unit.
14. Storage of data happens in the _____ component.

UNIT II

2.1 PERFORMANCE COMPARISON

The performance of CPU Scheduling algorithms depend on a variety of features including the probability distribution of service times of various processes, the efficiency of the scheduling and context switching mechanisms, and nature of the I/O demand and the performance of the I/O subsystem.

2.1.1 DETERMINISTIC MODEL

This method takes a particular predetermined work load and defines the performance of each algorithm for that work load. For example, assume that we have a workload – All the 5 processes arrive at time 0. The length of CPU burst time is given as follows

Process	CPU Burst time (in ms)
P1	6
P2	12
P3	1
P4	3
P5	4

Consider FCFS, SJF, and RR (Quantum = 1) scheduling algorithms for this process. The algorithm to give us the least Average wait time could be called the best one. In our example, the SJF algorithm would have the least waiting time and hence is the best one. This kind of algorithm execution is called Deterministic Modelling. It is simple and fast. It requires the exact number for input and its answers apply only to those cases.

2.1.2 QUEUING ANALYSIS

Queuing Analysis compares the scheduling algorithm by computing the .CPU utilization, average queue length, average wait time, etc when the arrival rates and service rates are known.

The Little's law is used for analysis here,

$$N = \lambda \times W$$

N – Average queue length

λ – Average arrival rate in the queue
W – Waiting time of the process

2.1.3 SIMULATORS

Simulations involve programming a model of the computer system. Software data structures represent the major components of the system. When the simulation executes statistics that indicate algorithm performance are gathered.

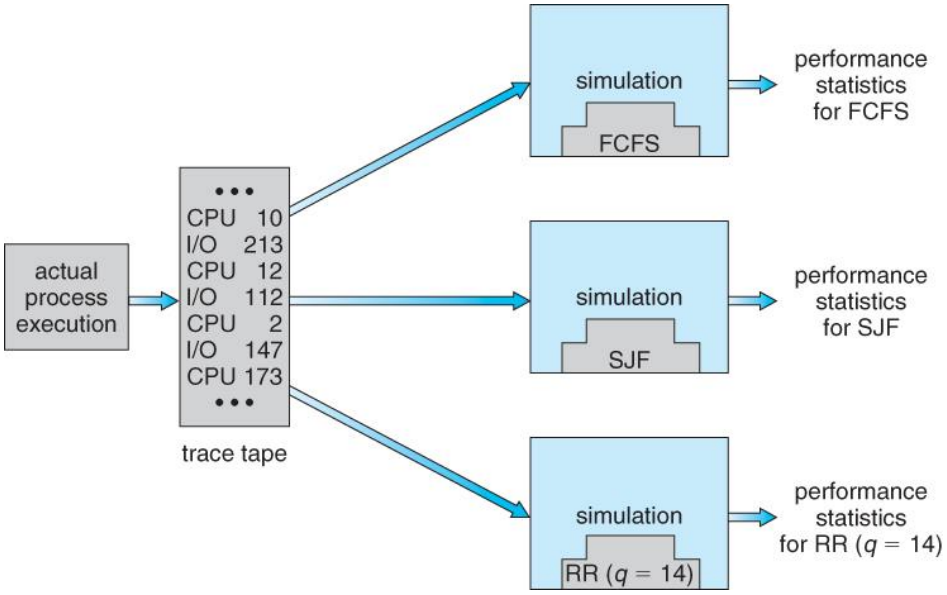


Fig 2.1 Evaluation of CPU Scheduling Algorithms using Simulator

2.2 DEADLOCK AND STARVATION

Deadlock:

A process in operating system uses resources in the following way.
1) Requests a resource
2) Use the resource
3) Releases the resource

In a multiprogramming environment, several processes may compete for a finite number of resources. A process requests resources; and if the resources are not available at that time, the process enters a waiting state. Sometimes, a waiting process is never again able to change state, because the resources it has requested are held by other waiting processes.

Deadlock	Starvation
The deadlock situation occurs when one of the processes got blocked.	Starvation is a situation where all the low priority processes got blocked, and the high priority processes execute.
Deadlock is an infinite process.	Starvation is a long waiting but not an infinite process.
Every Deadlock always has starvation.	Every starvation does n't necessarily have a deadlock.
Deadlock happens then Mutual exclusion, hold and wait. Here, preemption and circular wait do not occur simultaneously.	It happens due to uncontrolled priority and resource management

Deadlock is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process.

For example, in the below diagram, Process 1 is holding Resource 1 and waiting for resource 2 which is acquired by process 2, and process 2 is waiting for resource 1.

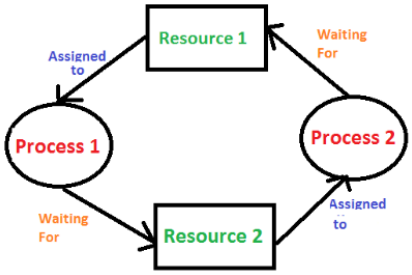


Fig 2.2 A Resource Allocation Graph

2.2.1 RESOURCE ALLOCATION GRAPH

Deadlocks can be described more precisely in terms of a directed graph called a system resource-allocation graph. So, resource allocation graph is explained to us what is the state of the system in terms of processes and resources. Like how many resources are available, how many are allocated and what is the request of each process. Everything can be represented in terms of the diagram. One of the advantages of having a diagram is, sometimes it is possible to see a deadlock directly by using RAG.

This graph consists of a set of vertices *V* and a set of edges *E*.

In RAG, vertices are two types –

1. **Process vertex** – Every process will be represented as a process vertex. Generally, the process will be represented with a circle. The set P = {P1, P2, ..., Pn} consists of all the active processes in the system.

2. **Resource vertex** – Every resource will be represented as a resource vertex. R = {R1, R2, …,Rn }is the set consisting of all resource types in the system. It is also of two types -

- **Single instance type resource** – It represents as a box, inside the box, there will be one dot. The numbers of dots indicate how many instances are present of each resource type.
- **Multi-resource instance type resource** – It also represents as a box, inside the box, there will be many dots present.

Now, coming to the edges of RAG - there are two types–

1. **Assign Edge** – If you already assign a resource to a process then it is called Assign edge.

2. **Request Edge** – It means in future the process might want some resource to complete the execution, this is called request edge.

A directed edge from process Pi to resource type Rj is denoted by Pi ⟶Rj; It signifies that process Pi has requested an instance of resource type Rj and is currently waiting for that resource. A directed edge from resource type Rj to process Pi is denoted by Rj ⟶ Pi; it signifies that an instance of resource type Rj has been allocated to process Pi. A directed edge Pi ⟶Rj is called a **request edge;** a directed edge Rj ⟶ Pi is called an **assignment edge**.

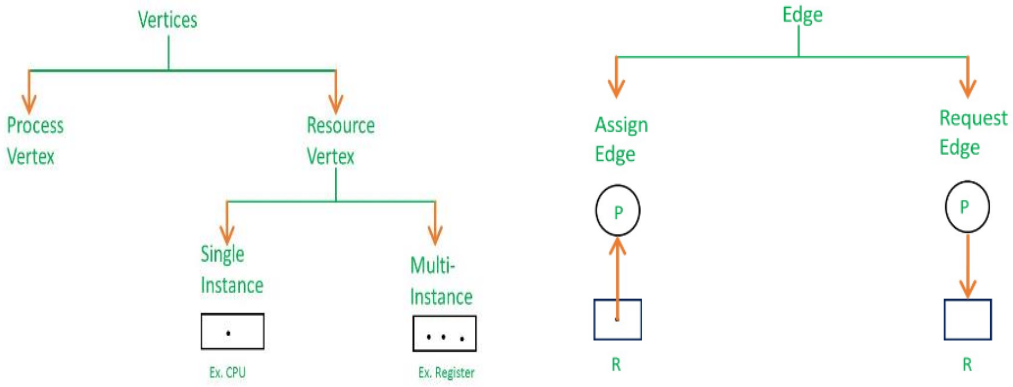


Fig 2.3 Vertices and Edges in RAG

The resource-allocation graph shown in Figure depicts the following situation.

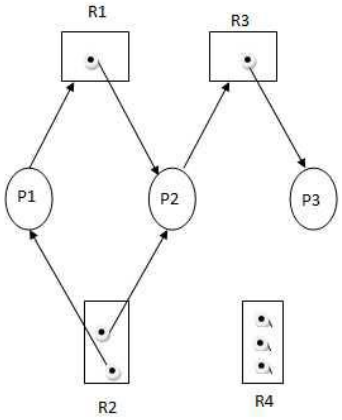


Fig. 2.4 Resource allocation graph

The sets P, R, and £:

- P={ P1, P2, P3}
- R={R1, R2, R3, R4}
- £ = {P1⟶R1, P2⟶R3,R1⟶P2, R2⟶P2, R2⟶P1, R3⟶P3 }

Resource instances:

- One instance of resource type R1.
- Two instances of resource type R2.
- One instance of resource type R3.
- Three instances of resource type R4.

Process states:

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

If the graph contains no cycles, then no process in the system is deadlocked. If the graph does contain a cycle, then a deadlock may exist. Suppose that process P3 requests an

instance of resource type R2. Since no resource instance is currently available, a request edge P3 ⟶ R2 is added to the graph. At this point, two minimal cycles exist in the system:

P1 ⟶ R1 ⟶ P2 ⟶ R3 ⟶ P3 ⟶ R2 ⟶ P1

P2 ⟶ R3 ⟶ P3 ⟶ R2 ⟶ P1

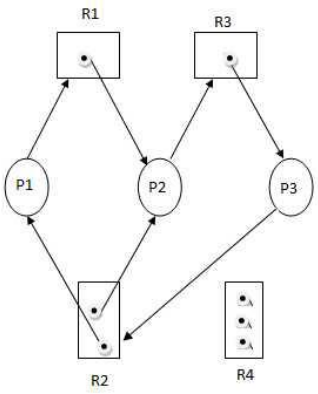


Fig 2.5 Resource-allocation graph with a deadlock

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

2.2.2 CONDITIONS FOR DEADLOCK

A deadlock situation can arise if the following four conditions hold simultaneously in a system:

1. Mutual exclusion: At least one resource must be held in a non-sharable mode; that is, only one process at a time can use the resource. If another process requests that resource, the requesting process must be delayed until the resource has been released.

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

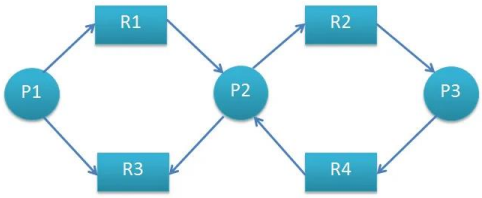


Fig 2.6 Deadlock by hold and wait

The processes P1, P2 and P3 each hold a resource, R1, R2, R3 and R4. In turn, each is also waiting for a different resource.

3. No pre-emption: Resources cannot be pre-empted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.

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

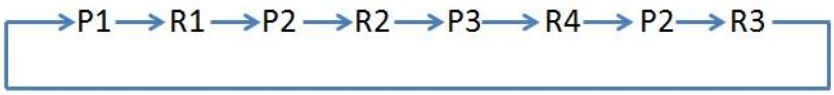


Fig 2.7 Circular wait Deadlock

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

2.2.3 DEADLOCK PREVENTION

For a deadlock to occur, each of the four necessary conditions must hold. By ensuring that at least one of these conditions cannot hold, we can prevent the occurrence of a deadlock. We elaborate on this approach by examining each of the four necessary conditions separately.

2.2.3.1 Mutual Exclusion

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

2.2.3.2 Hold and Wait

To ensure that the hold-and-wait condition never occurs in the system, we must guarantee that, whenever a process requests a resource, the process can procure the resource if, and only if, it does not hold any other resources. That is, before it can request any additional resources, however, it must release all the resources that it is currently allocated. This can be an expensive protocol and could consume a lot of time.

The next protocol that can be used requires each process to request and be allocated all its resources before it begins execution. This is also difficult as a process may require a lot of resources to begin its execution.

To illustrate the difference between these two protocols, we consider a process that copies data from a DVD drive to a file on disk, sorts the file, and then prints the results to a printer. If all resources must be requested at the beginning of the process, then the process must initially request the DVD drive, disk file, and printer. It will hold the printer for its entire execution, even though it needs the printer only at the end. The second method allows the process to request initially only the DVD drive and disk file. It copies from the DVD

drive to the disk and then releases both the DVD drive and the disk file. The process must then again request the disk file and the printer. After copying the disk file to the printer, it releases these two resources and terminates.

Both these protocols have two main disadvantages. First, resource utilization may be low, since resources may be allocated but unused for a long period. Second, starvation is possible. A process that needs several popular resources may have to wait indefinitely, because at least one of the resources that it needs is always allocated to some other process.

2.2.3.3 No Pre-emption

The third necessary condition for deadlocks is that there be no pre-emption of resources that have already been allocated. To ensure that this condition does not hold, we can use the following protocol. If a process is holding some resources and requests another resource that cannot be immediately allocated to it (that is, the process must wait), then all resources currently being held are pre-empted. In other words, these resources are implicitly released. The pre-empted resources are added to the list of available resources for which the other process is waiting. The process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

Alternatively, if a process requests some resources, we first check whether they are available. If they are, we allocate them. If they are not, we check whether they are allocated to some other process that is waiting for additional resources. If so, we pre-empt the desired resources from the waiting process and allocate them to the requesting process.

If the resources are neither available nor held by a waiting process, the requesting process must wait. While it is waiting, some of its resources may be pre-empted, but only if another process requests them. A process can be restarted only when it is allocated the new resources it is requesting and recovers any resources that were pre-empted while it was waiting.

2.2.3.4 Circular Wait

The fourth and final condition for deadlocks is the circular-wait condition. One way to ensure that this condition never holds is to give some particular ordering of all resource types and to require that each process requests resources in an increasing order of enumeration.

To illustrate, we let R = {R1, R2,..., Rm} be the set of resource types. We assign to each resource type a unique integer number, which, allows us to compare two resources and to determine whether one precedes another in our ordering. We define a one-to-one function F: R → N, where N is the set of natural numbers. For example, if the set of resource types R includes tape drives, disk drives, and printers, then the function F might be defined as follows:

F(tape drive) = 1
F(disk drive) = 5
F(printer) = 12

We can now consider the following protocol to prevent deadlocks: Each process can request resources only in an increasing order of enumeration. That is, a process can initially

request any number of instances of a resource type Ri. After that, the process can request instances of resource type Rj if and only if F(Rj) > F(Ri). If several instances of the same resource type are needed, a single request for all of them must be issued.

For example, using the function defined previously, a process that wants to use the tape drive and printer at the same time must first request the tape drive and then request the printer. Alternatively, we can require that, whenever a process requests an instance of resource type Rj, it has released any resources Ri such that F(Ri) ≥ F(Rj). If these two protocols are used, then the circular-wait condition cannot hold.

2.2.4 DEADLOCK DETECTION

When a deadlock situation occurs, the system must provide an algorithm that examines the state of the system to determine whether a deadlock has occurred. We can detect the deadlocks using wait for graph for single instance resource type and detect using detection algorithm for multiple instances of resource type.

2.2.4.1 Single Instance of Each Resource Type

Single instance of resource type means, the system consisting of only one resource of one type. This type of dead lock can be detected with the help of wait for graph. Wait for a graph is a graph which is derived from Resource allocation graph. It consisting of only processes as vertices. We obtain this graph from the resource-allocation graph by removing the resource nodes and collapsing the appropriate edges.

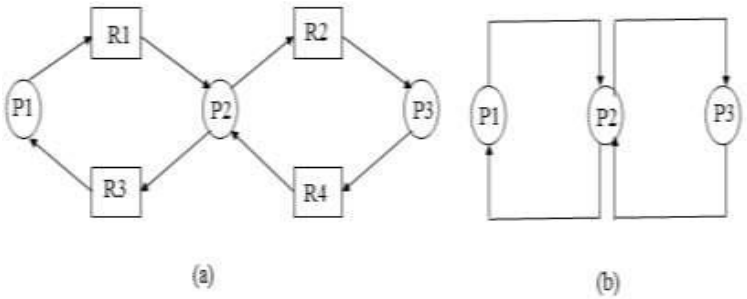


Fig 3.4 (a) Resource-allocation graph (b) Corresponding wait-for graph

A deadlock exists in the system if and only if the wait-for graph contains a cycle. To detect deadlocks, the system needs to maintain the wait-for graph and periodically invoke an algorithm that searches for a cycle in the graph. An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph.

2.2.4.2 Several Instances of a Resource Type

The wait-for graph scheme is not applicable to a resource allocation system with multiple instances of each resource type. For multiple instances of each resource a deadlock-detection algorithm is applicable. This algorithm requires an order of $m \times n^2$ operations to detect whether the system is in a deadlocked state. Following are several time varying data structures

- **Available**: A vector of length m indicates the number of available resources of each type.

- **Allocation**: An n × m matrix defines the number of resources of each type currently allocated to each process.
- **Request**: An n × m matrix indicates the current request of each process. If Request[i][j] = k, then process Pi is requesting k more instances of resource type Rj.

To simplify notation, we again treat the rows in the matrices Allocation and Request as vectors; we refer to them as Allocation‘i’ and Request‘i’, respectively

Algorithm:

Step 1: Let Work and Finish be vectors of length m and n, respectively.
Initialize
Work = Available.
For i = 0, 1, …, n−1,
If Allocationi ≠ 0, then Finish[i] = false;
otherwise, Finish[i] = true.

Step 2: Find an index i such that both
a. Finish[i] = false
b. Requesti ≤ Work
If no such i exists, go to step 4.

Step 3: Work = Work + Allocationi
Finish[i] = true
Go to step 2.

Step 4: If Finish[i] == false, for some i, 0 ≤ i < n, then the system is in a deadlocked state. Moreover, if Finish[i] == false, then process Pi is deadlocked.

2.2.5 RECOVERY FROM DEADLOCK

There are two options for breaking a deadlock. One is simply to abort one or more processes to break the circular wait. The other is to pre-empt some resources from one or more of the deadlocked processes.

2.2.5.1 Process Termination

There are two methods for terminating process. We use one of two methods. In both methods, the system reclaims all resources allocated to the terminated processes.

- **Abort all deadlocked processes**: Release all deadlocked processes and begin the allocating from the first. This method clearly will break the deadlock cycle, but at great expense.
- **Abort one process at a time until the deadlock cycle is eliminated**: Here, one of the processes in the deadlock is aborted and the deadlock is checked, if it remains, the procedure continues for another process in deadlock, until the deadlock is dissolved. This method incurs considerable overhead, since, after each process is aborted, a deadlock-detection algorithm must be invoked to determine whether any processes are still deadlocked. Though expensive, this process is more cost effective than the above alternative.

2.2.5.2 Resource Pre-emption

To eliminate deadlocks using resource pre-emption, we successively pre-empt some resources from processes and give these resources to other processes until the deadlock cycle is broken. There are three methods to eliminate deadlocks using resource pre-emption.

- **Selecting a victim**: Here, select resources and processes of the processor are to be pre-empted. The cost factors may include the number of resources a deadlocked process is holding and the amount of time the process has consumed during its execution.
- **Rollback**: If we pre-empt a resource from a process, what should be done with that process? Clearly, it cannot continue with its normal execution; it is missing some needed resource. We must roll back the process to some safe state and restart it from that state. Since, in general, it is difficult to determine what a safe state is. The simplest solution is a total rollback - Abort the process and then restart it. Although it is more effective to roll back the process only as far as necessary to break the deadlock, this method requires the system to keep more information about the state of all running processes.
- **Starvation**: How do we ensure that starvation will not occur? That is, how can we guarantee that resources will not always be pre-empted from the same process? In a system where victim selection is based primarily on cost factors, it may happen that the same process is always picked as a victim. As a result, this process never completes its designated task, a starvation situation that needs to be dealt with in any practical system. Clearly, we must ensure that a process can be picked as a victim only a (small) finite number of times. The most common solution is to include the number of rollbacks in the cost factor.

2.3 Questions

1. What is Deadlock?
2. Deadlock Detection in OS
3. Deadlock Prevention in OS
4. Deadlock Recover Algorithms
5. Difference Between Starvation and Deadlock

UNIT III

3.1 MEMORY MANAGEMENT

In uni-programming system, the main memory is divided into two parts, one part is for operating system and another part is for currently executing job. Consider the following figure.

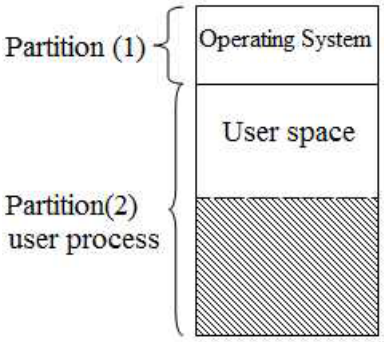


Fig 3.1 Main memory partition

Partition 1 is used operating system and partition 2 is used to store the user process. In partition 2 some part of memory is wasted, it is indicated by blacked lines in figure. In multiprogramming environment the user space is divided in to number of partitions. Each partition is for one process. The task of sub division is carried out dynamically by the operating system; this task is known as "Memory Management". The efficient memory management is possible with multiprogramming.

3.1.1 LOGICAL vs PHYSICAL ADDRESS SPACE

An address generated by the CPU is commonly referred to as a logical address, where as an address seen by the memory unit, that is, the one loaded into the memory-address register of the memory, is commonly referred to as a physical address.

The compile-time and load-time address-binding methods generate identical logical and physical addresses. However, the execution-time address-binding scheme results in differing logical and physical addresses. In this case, we usually refer to the logical address as a virtual address. We use logical address and virtual address interchangeably in this text.

The set of all logical addresses generated by a program is a logical-address space; the set of all physical addresses corresponding to these logical addresses is a physical-address space. Thus, in the execution-time address binding scheme, the logical- and physical-address spaces differ. The run-time mapping from virtual to physical addresses is done by a hardware device called the memory-management unit (MMU).

For example, J1 is a program, written by the user, the size of the program is 100KB. But program loaded in main memory from 2100 to 2200 KB. This actual loaded address in main memory is called physical address.

The set of all logical address is generated by a program is referred to as a "Logical address space". In our example from 0 to 100KB is the logical address space and from 2100 to 2200KB in the physical address space.

Physical address space = Logical address space + Contents of relocation register/ Base value

i.e., 2200 = 100 + 2100

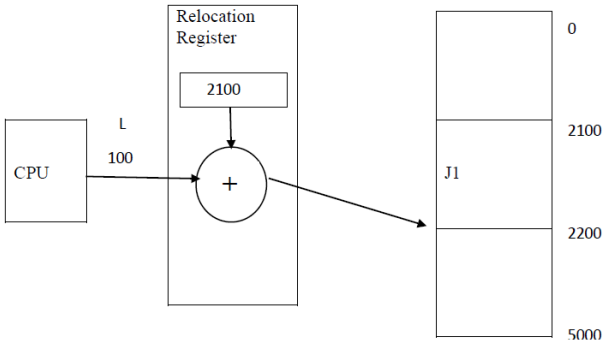


Fig 3.2 Dynamic relocation of data using a Relocation register

The base register is now called a relocation register. The value in the relocation register is added to every address generated by a user process at the time it is sent to memory. For example, the base is at 2100, then the attempt by the user to address location 0 is dynamically related to location 2100. An access location 100 is mapped to location 2200.

We now have two different types of addresses: logical addresses (in the range 0 to max) and physical addresses (in the range R + 0 to R + max for a base value R). The user generates only logical addresses and thinks that the process runs in locations 0 to max. The user program supplies logical addresses; these logical addresses must be mapped to physical addresses before they are used. The value of the relocation register is added to every address generated by a user process at the time it is sent to the memory. The concept of a logical-address space that is bound to a separate physical-address space is central to proper memory management

3.1.2 SWAPPING

Swapping is used to increase main memory utilization. For example main memory consisting of 15 processes, assume that it is the maximum capacity of memory to hold the processes. The CPU currently executing the process no: 14 in the middle of the execution the process 14 needs I/O. then the CPU switches to the another job and process 14 is moved to a disk and the another process is loaded in to the main memory in place of process 14.

When the process 14 is completed its I/O operation then the process 14 is moved to the main memory from disk. Switching process from main memory to disk is known as swap out‖ and switching from disk to main memory is called swap in. This type of mechanism is said to be Swapping‖. We can achieve the efficient memory utilization with swapping.

Swapping requires 'Backing store'. The backing store is commonly a fast disk. It must be large enough to accommodate the copies of all process images for all users. When a

process is swapped out, its executable image is copied into backing store. When it is swapped in, it is copied into the main memory at new block allocated by the memory manager.

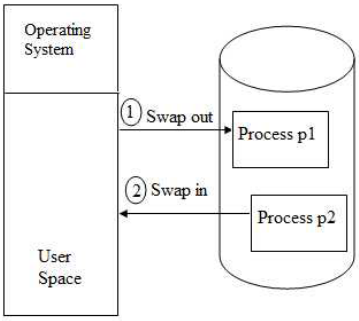


Fig 3.3 Swapping in and out of Main memory (left) and Secondary memory (right)

3.1.3 MEMORY MANAGEMENT REQUIREMENT

Memory management keeps track of the status of each memory location, whether it is allocated or free. It allocates the memory dynamically to the programs at their request and frees it for reuse when it is no longer needed. Memory management meant to satisfy some requirements that we should keep in mind.

- Relocating
- Protection
- Sharing
- Logical Organisation
- Physical Organization

3.1.3.1 Relocating

It is the mechanism of converting logical addresses into physical addresses. The address generated by CPU is the logical address and the address generated by the memory manager is the physical address. Relocation is necessary at the time of swap in a process from one backing store to the main memory.

Physical address= Contents of Relocation register + logical address

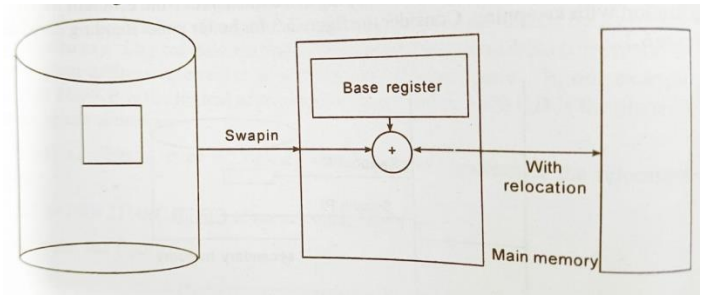


Fig3.4 Relocating during Swap

3.1.3.2 Protection

Protecting the operating system from user processes and protecting user processes from one another. We can provide this protection by using a relocation register, with a limit register. The relocation register contains the value of the smallest physical address; the limit

register contains the range of logical addresses (for example, relocation = 100040 and limit = 74600). With relocation and limit registers, each logical address must be less than the limit register; the MMU maps the logical address *dynamically* by adding the value in the relocation register. This mapped address is sent to memory.

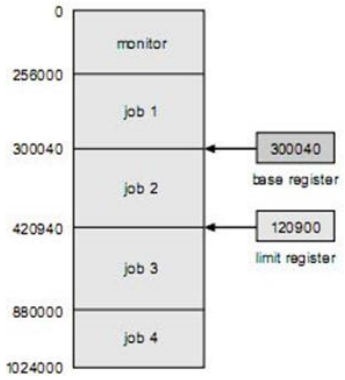


Fig 3.5 Protection – Main Memory

When the CPU scheduler selects a process for execution, the dispatcher loads the relocation and limit registers with the correct values as part of the context switch. Because every address generated by the CPU is checked against these registers, we can protect both the operating system and the other users' programs and data from being modified by this running process

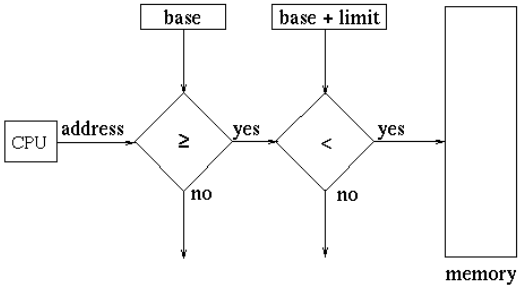


Fig 3.6 Memory protection with base and limit

3.1.3.3 Sharing

A protection mechanism must have to allow several processes to access the same portion of main memory. Allowing each processes access to the same copy of the program rather than having their separate copy has an advantage. For example, multiple processes may use the same system file and it is natural to load one copy of the file in main memory and let it be shared by those processes. It is the task of Memory management to allow controlled access to the shared areas of memory without compromising the protection. Mechanisms are used to support relocation-supported sharing capabilities.

For example, if 3 users are using the word processor the processor is loaded only once and this memory space is shared by all 3 users. Making a copy of the word program thrice would be a waste.

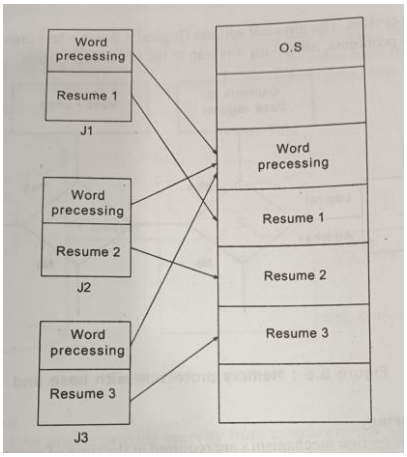


Fig 3.7 Shared memory Example

3.1.3.4 Logical Organisation

Main memory is organized as linear or it can be a one-dimensional address space which consists of a sequence of bytes or words. Most of the programs can be organized into modules, some of those are unmodifiable (read-only, execute only) and some of those contain data that can be modified. To effectively deal with a user program, the operating system and computer hardware must support a basic module to provide the required protection and sharing. It has the following advantages:

- Modules are written and compiled independently and all the references from one module to another module are resolved by `the system at run time.
- Different modules are provided with different degrees of protection.
- There are mechanisms by which modules can be shared among processes. Sharing can be provided on a module level that lets the user specify the sharing that is desired.

3.1.3.5 Physical Organisation

The structure of computer memory has two levels referred to as main memory and secondary memory. Main memory is relatively very fast and costly as compared to the secondary memory. Main memory is volatile. Thus secondary memory is provided for storage of data on a long-term basis while the main memory holds currently used programs. The major system concern between main memory and secondary memory is the flow of information and it is impractical for programmers to understand this for two reasons:

- The programmer may engage in a practice known as overlaying when the main memory available for a program and its data may be insufficient. It allows different modules to be assigned to the same region of memory. One disadvantage is that it is time-consuming for the programmer.
- In a multiprogramming environment, the programmer does not know how much space will be available at the time of coding and where that space will be located inside the memory.

3.1.4 DYNAMIC LOADING and DYNAMIC LINKING

Loading means to load the program or module from the secondary storage device to the main memory. Loadings are two types:

1. Compile-time loading –All the routines are loaded in the main memory at the time of compilation. It is also known as Static loading
2. Runtime loading – The routines are loaded in the main memory at the runtime or execution time that is called Dynamic Loading**.**

Consider a simple C program,

```
#include <stdio.h>
main() {
printf("Welcome to world of OS");
}
```

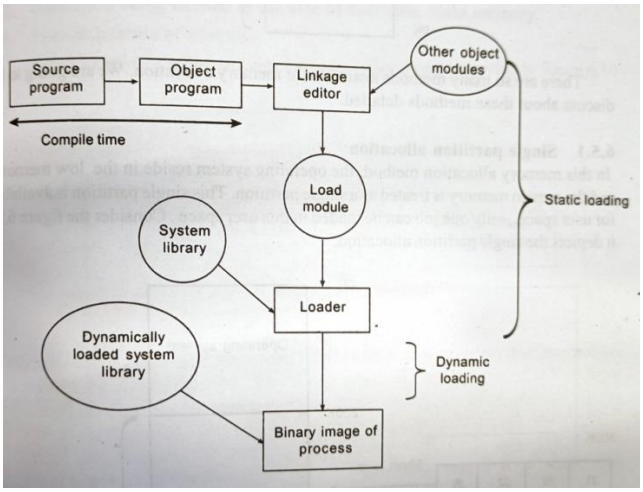


Fig 3.8 Step by step of user program execution

Suppose the program occupies 1 kB in secondary storage. It would occupy more space if the entire stdio is loaded in main memory. Instead, with dynamic loading, a routine s not loaded until it is called. So, an unused routine/ function/ procedure is not loaded in the memory at all.

The linking of the library files at the execution time is called **Dynamic Linking**. Otherwise, Dynamic Linking is the linking is postponed until execution time. Linking of library files before execution is called static linking. In dynamic linking, linking is postponed until the execution time.

Dynamic Loading	Dynamic Linking
Dynamic loading is the process of loading system library or routine at run-time.	Dynamic linking is the process of linking the dependent library or routine at run-time.
Dynamic loading is not supported by OS	Dynamic linking is supported by OS

3.1.5 MEMORY ALLOCATION METHOD

The main memory contains both the operating system and the users' processes. OS is either placed in the low memory or high memory. It is usually in the placed in the low memory.

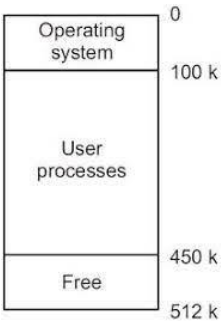


Fig 3.9 OS placed in low memory

3.1.5.1 Single Partition Allocation

In this method, the operating system resides in the lower part of the memory, and the remaining memory is treated as a single partition. This single partition is available for the user's space. Only a single job can be loaded in this user space at time. The short term scheduler selects a job from ready queue for execution, and the dispatcher loads that job in main memory. The main memory consists of only one process at time, because the user space treated as a single partition.

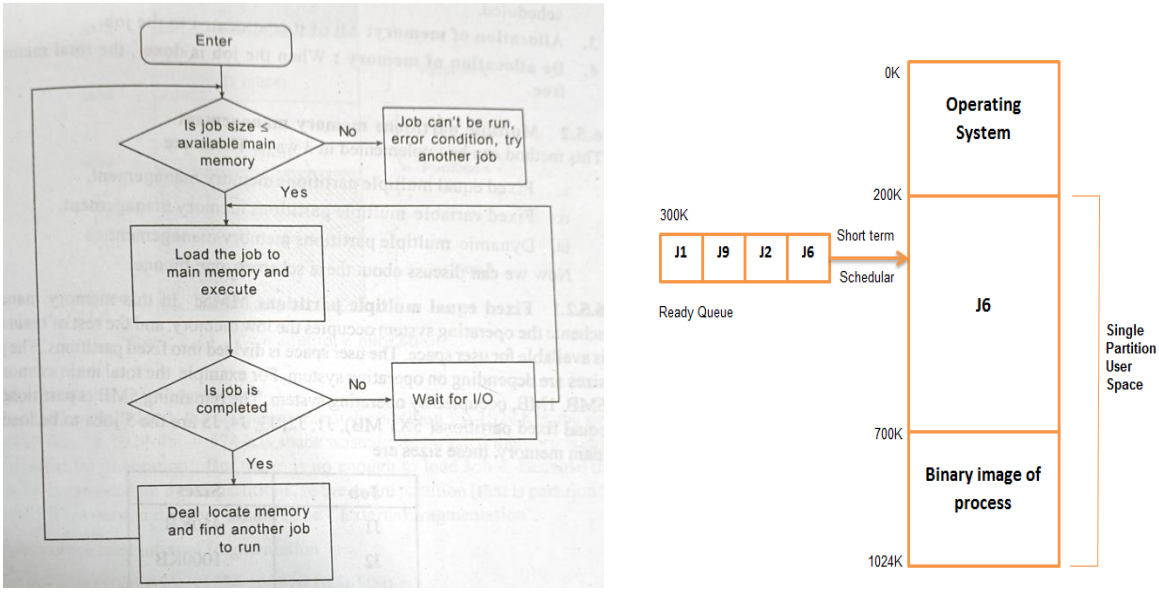


Fig 3.10 Flow chart for job allocation and Memory allocation in single partition

Memory Management function:

- Keeping Tracking of memory: Total memory allocated to the job
- Determining factor of memory policy: The job gets all memory when scheduled
- Allocation of memory: All of the memory is allocated to the job
- De-allocation of memory: When job is done, all of the memory is free

Advantages:

- Simple method. No expertise needed to use the system.

Disadvantages:

- Poor utilization of processors
- User job is restricted to the size of available main memory
- Poor utilization of memory

3.1.5.2 Multiple Partitions Memory Management

This method can be implemented in three ways

a. Fixed equal multiple partitioning
b. Fixed variable multiple partitioning
c. Dynamic multiple partitioning

Fixed equal multiple partitioning:

In this scheme the operating system resides in the low memory and rest of main memory is used as user space. The user space is divided in to fixed partitions. The size of these partitions is depending up on the operating system. For example the total main memory size is 6MB; 1MB is occupied by the operating system. The remaining 5MB is partitioned in to 5 equal fixed partitions of 1MB each. P1, P2, P3, P4, P5 are the 5 jobs to be loaded in the main memory, there size is given in the table below

Job	size
P1	450kb
P2	1000kb
P3	1024kb
P4	1500kb
P5	500kb

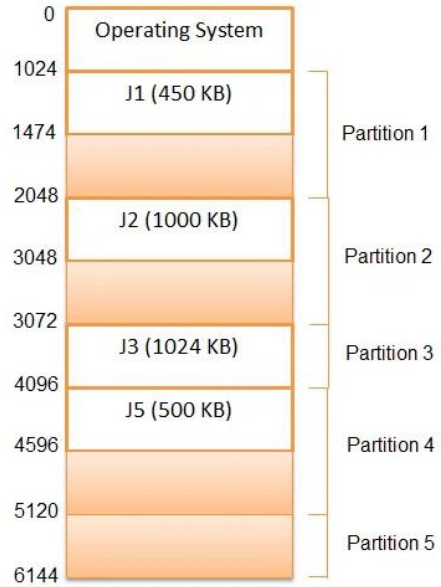


Fig 3.11Memory allocation with an example

Internal and external fragmentation:

Process P1 is loaded into partition1. The maximum size of partition1 is 1024kb, the size of P1 is 450kb. So 1024 – 450 = 574kb space is wasted, this wasted memory is said to be Internal fragmentation. But there is no enough space to load process P4, because the size of process P4 is greater than all the partitions, so the entire partition (partition5) is wasted. This wasted memory is said to be External fragmentation. Therefore the total internal fragmentation is

= (1024 - 450) + (1024 - 1000) + (1024 - 500) = 574 + 24 + 524 = 1122kb

The external fragmentation is 1024 kb.

A part of memory wasted within a partition is called internal fragmentation and the wastage of an entire partition is called external fragmentation.

Advantages:

- It supports multiprogramming.
- Effective utilization of the processor and I/O devices is possible.

- Requires no special costly hardware
- Simple and easy to implement

Disadvantage:

- It suffers from internal and external fragmentation.
- The single free area might not be large enough for a partition
- Requires more memory than single partition method
- A job's partition size depends on the size of the physical memory.

Fixed variable multiple partitioning:

In this method the user's space of main memory is divided into number of partitions, but partitions size are different in length. The operating system keeps a table indicating which partitions of memory are available and which are occupied. When a process arrives and needs memory, we search for a partition large enough for this process. If we find the space large enough to fit the process, allocate the partition to that process.

For example, assume that we have 4000kb of main memory available, and operating system occupies 500kb. The remaining 3500kb of memory is used for user's processes.

Job Queue		
Job	Size	Arrival Time (ms)
J1	825 kb	10
J2	600 kb	5
J3	1200 kb	20
J4	450 kb	30
J5	650 kb	15

Partitions	
Partition	Size
P1	700 kb
P2	400 kb
P3	525 kb
P4	900 kb
P5	350 kb

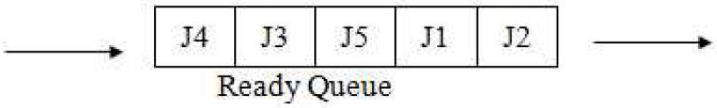


Fig 3.12 Scheduling Example

- Here we use first-fit strategy to illustrate the problem. Out of 5 jobs J2 arrives first, the size of J2 is 600KB,. Searching can be started from low memory to high memory and first partition which is big enough is allocated. Here P1 is first partition which is big enough for J2, so load the J2 in P1.
- J1 is next job in the ready queue. The size is 825KB; P4 is the first partition that is big enough, so load the J1 in to P4.
- J5 arrives next, the size is 650KB, and there is no large enough partition to load that job, so J5 has to wait until enough partition is available.
- J3 arrives next, the size is 1200KB. There is no large enough space to load this one also. J4 arrives last, the size is 450KB, and partition P3 is large enough to load this process. So load J4 in to P3.

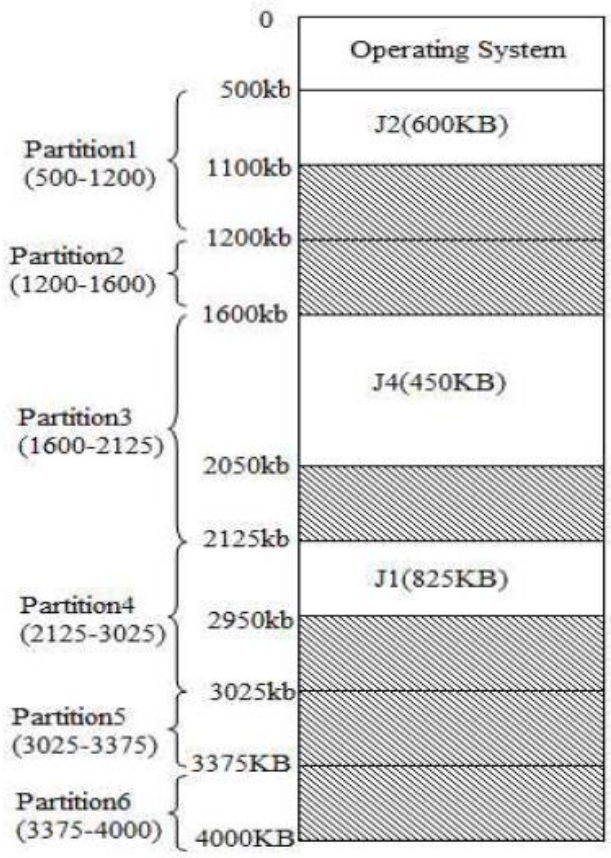


Fig 3.13 Memory Allocation of the example

Partitions P2,P5,P6 are totally free, there is no processes in these partitions. This wasted memory is said to be external fragmentation. The total external fragmentation is 1375, (400+350+625).

The total internal fragmentation is (700-600) + (525-450) + (900-825) =250.

There are three strategies used to allocate memory in this scheme

- **First-Fit:** It allocates the first partition that is big enough. Searching can start either from low memory or high memory. We can stop searching as soon as we find a free partition that is large enough.
- **Best-Fit:** It allocates the smallest partition that is big enough. Searching can be started from either end of memory, Searches entire memory, and allocates the smallest partition that is big enough for the process.
- **Worst-Fit:** Search the entire memory and selects the partition which is largest of all.

Advantages:

- It supports multiprogramming.
- Effective utilization of the processor and memory.
- Simple and easy to implement

Disadvantage:

- It suffers from internal and external fragmentation.
- Possible large external fragmentation.

Dynamic multiple partitioning

In this method partitions are created dynamically, so that each process is loaded in to partition of exactly the same size at that process. Here the entire user space is treated as a single partition that is big hole. The boundaries of partitions are dynamically changed. These boundaries are depending on the size of processes. Consider following example.

Job Queue		
Job	Size	Arrival Time (ms)
J1	825 kb	10
J2	600 kb	5
J3	1200 kb	20
J4	450 kb	30
J5	650 kb	15

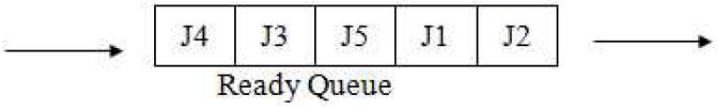


Fig 3.14 Job Queue

Job J2 arrives first, so load the J2 into memory first. Next J1 arrives. Load the J1 in to memory next. Then load J5,J3 and J4 into the memory. Consider following figure 3.14 for better understanding

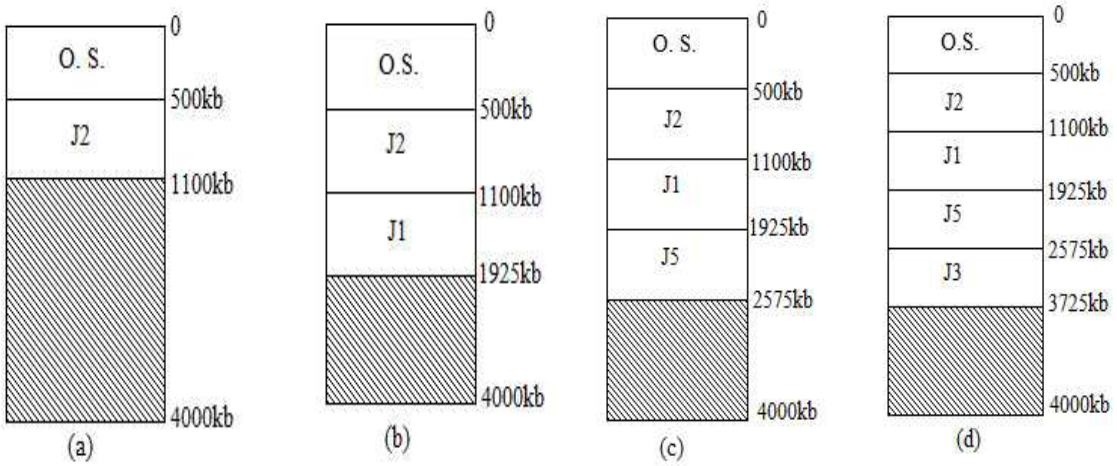


Fig 3.15 Dynamic Memory Allocation

In figure (a), (b), (c), (d) jobs J2, J1, J5, and J3 are loaded. The last job is J4, the size of J4 is 450kb, but the available memory is 225kb, which is not enough to load J4, so the job J4 has to wait until the memory is free. Assume that after some time J5 has finished and it releases its memory. Then the available memory becomes 225 + 650 = 875kb. This memory is enough to load J4. Consider the following figure 3.14 (e) and (f).

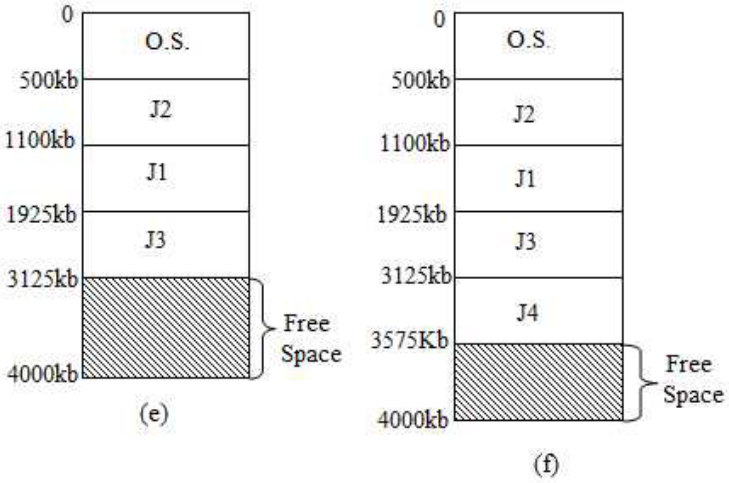


Fig 3.16 Contd., Memory Allocation

Advantages:

- In this method partitions are changed dynamically, so it does not suffer from internal fragmentation.
- Efficient memory and processor utilization are possible.

Disadvantage:

- It suffers from external fragmentation.

3.1.5.3 Compaction

Compaction is a method of collecting all the free spaces together in one block, this block can be allotted to some other job. For example consider the example of Fixed variable multiple partitioning method. The total internal fragmentation is (100+75+75=250Kb). The total external fragmentation is (400+350+625=1375Kb). Collect the internal and external fragmentation together in one block (250+1375=1625Kb). This type of mechanism is said to be compaction. Now the compacted memory is 1625Kb.

Now the scheduler can load job J3 (1200Kb) in compacted memory. Thus efficient memory utilization can be possible using compaction.

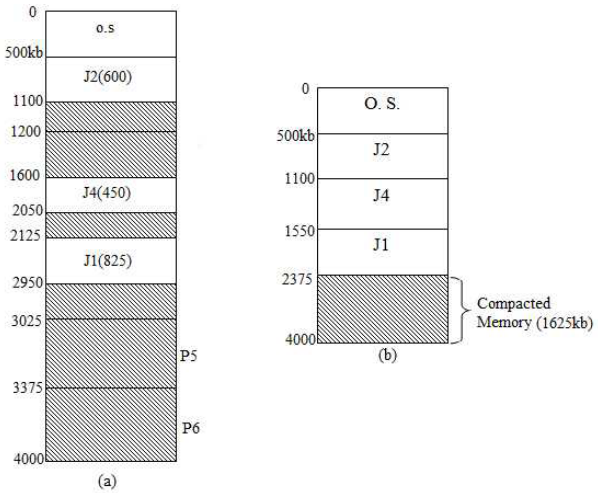


Fig 3.17 Compaction

3.1.5.4 Paging

The single and multiple partitioning methods support continuous memory allocation, the entire process loaded in partition. In paging the process is divided in number of small parts, these are loaded in to elsewhere in the main memory.

The physical memory is divided in to fixed sized blocks called frames; the logical address space (user Process) is divided in to fixed sized blocks called pages. The page size and the frame size must be equal. The size of page or frame is depending on the operating system. Generally the page size is 4KB.

In this method operating system maintain a data structure, called page table, it is used for mapping purpose. The page table specifies some useful information, it tells which frames are allocated and which frames are available, and how many total frames are there and so on. The general page table consisting of two fields, one is page number and other one if frame number. Each operating system has its own method for storing page table.

Every address generated by the CPU is divided into two parts; one is page number and second is page offset or displacement. The page number is used index in page table. Consider the following figure, the logical address space that is CPU generated address space is divided into pages, each page having the page number (P) and displacement (D) . The pages are loaded in to available free frames in the physical memory.

Page table contain the base address of each page in physical memory, that address is combined with offset to find the physical address of the page. The mapping between the page number and frame number is done by page map table. The page map table specifies which page is loaded in which frame, displacement is common. For better understanding consider the following example. There are two jobs in the ready queue, the size of job1 is 16kb and job 2 is 24kb. The page size is 4kb. The available main memory is 72kb i. e. 18 frames. So job 1 is divided in to 4 pages and job 2 is divided in to 6 pages. Each process maintains a program table. Consider the following figure for better understanding.

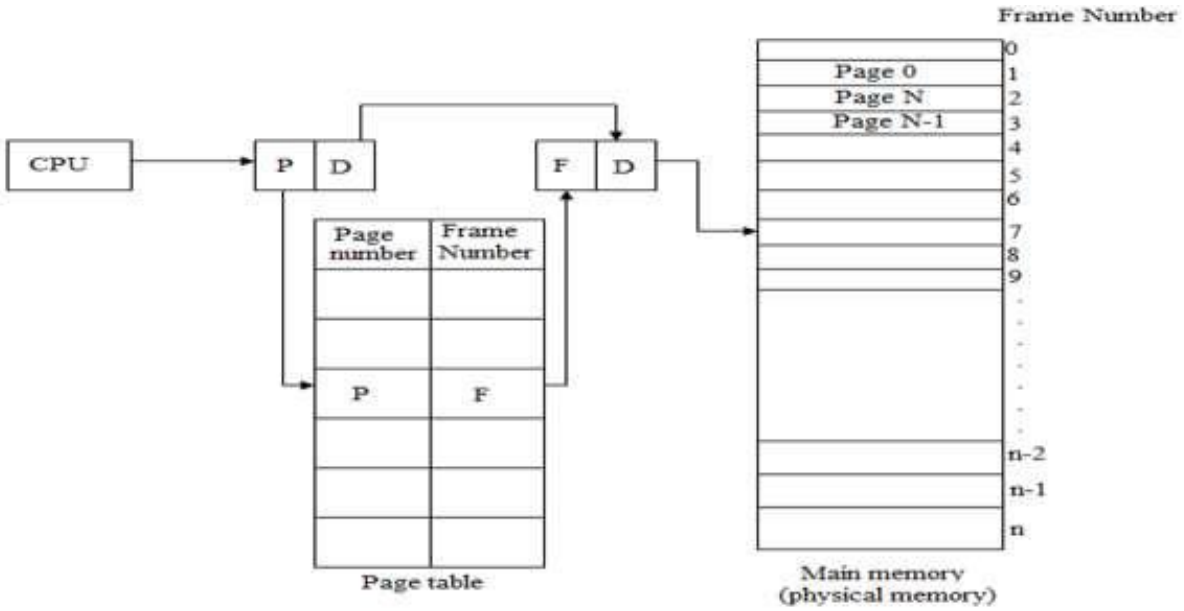


Fig 3.18 Structure of paging scheme

Four pages of job 1 are loaded in different locations in main memory. The O.S. provides a page table for each process. The page table specifies the location in main memory. The capacity of main memory in this example is 18 frames, and available jobs requires only 10 frames, so remaining 8 frames are free. These free frames can be used for some other jobs.

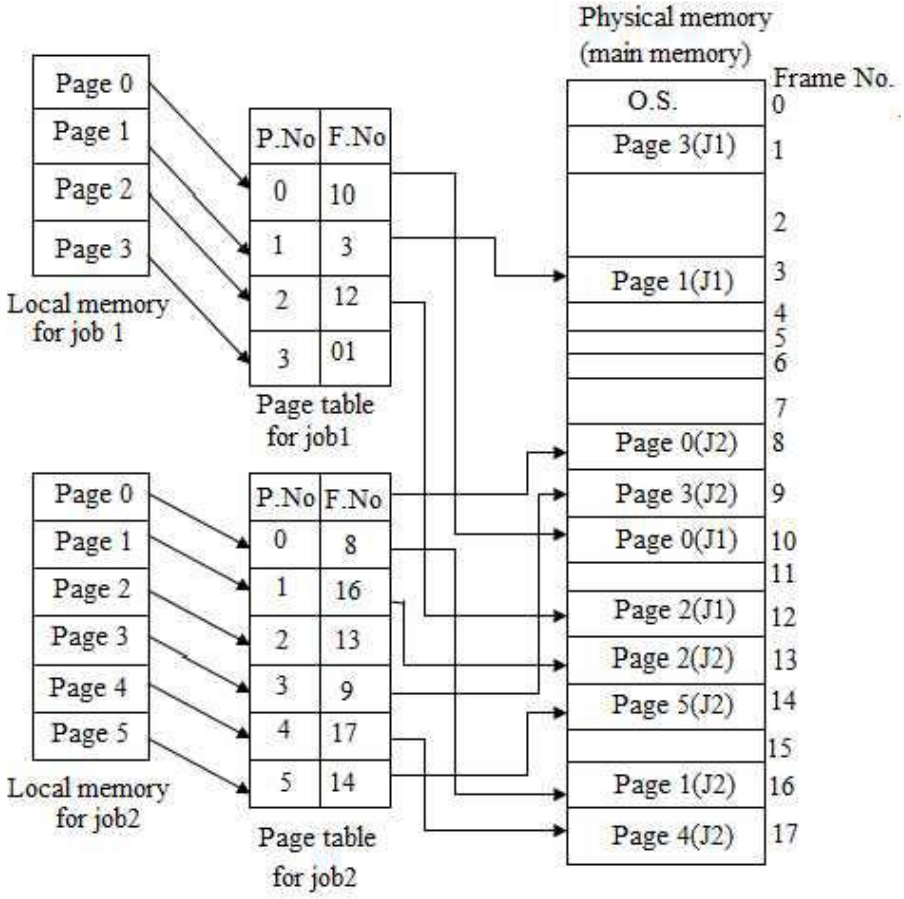


Fig 3.19 Paging – example

Advantages:

- Paging supports the time sharing system.
- It does not effect from fragmentation.
- It supports virtual memory.
- Sharing of common code is possible

Disadvantages:

- Paging may suffer page breaks. For example consider a job with the logical address space 17kb, the page size is 4kb. So this job requires 5 frames. The last frame i.e. fifth frame requires only 1kb of memory, so the remaining 3kb is wasted. It is said to be page breaks.
- If the number of pages is large, then it is difficult to maintain the page table.

Shared Pages:

In multiprogramming environment, it is possible to share the common code by number of processes at a time, instead of maintaining the number of copies of same code. The logical address is divided into pages, these pages can be shared by number of processes at a time. The pages which are shared by number of processes are called shared pages.

For example, consider a multiprogramming environment with 10 users. Out of 10 users, 3 users wish to execute a text editor; they want to take their bio-data in text editor. Assume that text editor requires 150kb and user bio-data occupies 50kb of data space. So they would need 3*(150+50)=600kb. But in shared paging the text editor is shared by all the users‘ jobs, so it requires 150kb and user files requires 50*3=150kb. Therefore (150+150) =300kb enough to manage these three jobs instead of 600kb. Thus shared paging saves 300kb of space.

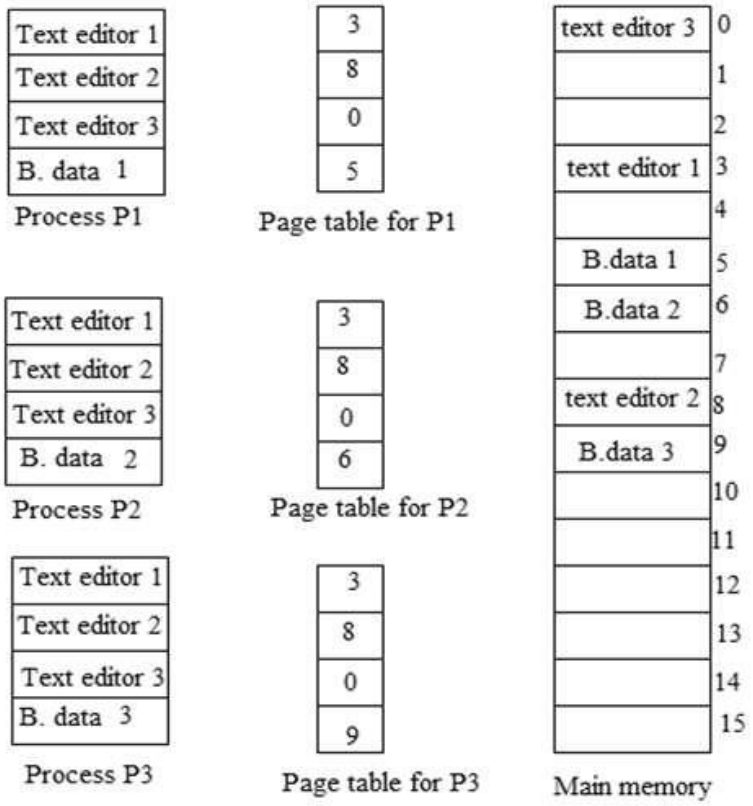


Fig 3.20 Shared Paging

The frame size and frame size is 50kb. So 3 frames are required for text editor and 3 frames are required for user files. Each process (P1, P2, P3) has a page table, the page table shows the frame numbers, the first 3 frame numbers in each page table i.e. 3,8,0 are common. It means the three processes shared the three pages. The main advantage of shared pages is efficient memory utilization is possible.

3.1.5.5 Segmentation

In segmentation the instructions are logically grouped, such as subroutines, arrays or other data areas. Every program is a collection of these segments. Segmentation is the technique for managing these segments. For example consider following figure.

Each segment has a name and a length. The address of the segment specifies both segment name and the offset within the segment. For example the length of the segment Main is 100kb. Main is the name of the segment. The operating system searches the entire main memory for free space to load a segment. This mapping is done by segment table. The segment table is a table having two entries segment Base and segment Limit

The segment base contains the starting physical address where the segment resides in memory. The segment limit specifies the length of the segment. Following figure shows the basic hardware for segmentation. The logical address consists of two parts: a segment number (s), and offset or displacement in to that segment (d). The segment number (s) is used as an index in to the segment table.

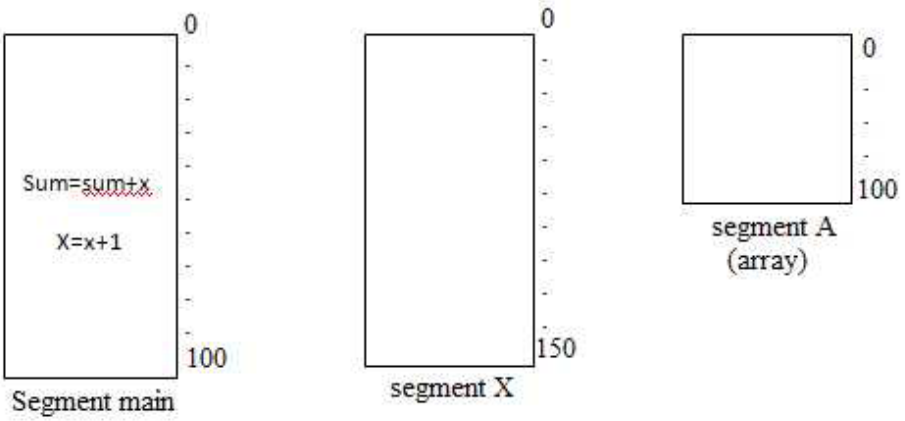


Fig 3.21 Segmented address space

For example consider the following figure in which the logical address space is (a job) is divided in to four segments, numbered from 0 to 3. Each segment has an entry in the segment table. The limit specifies the size of the segment and the base specifies the starting address of the segment. Here segment _0' is loaded in to main memory from 1500kb to 2500kb, so 1500kb is base and 2500-1500=1000kb is the limit.

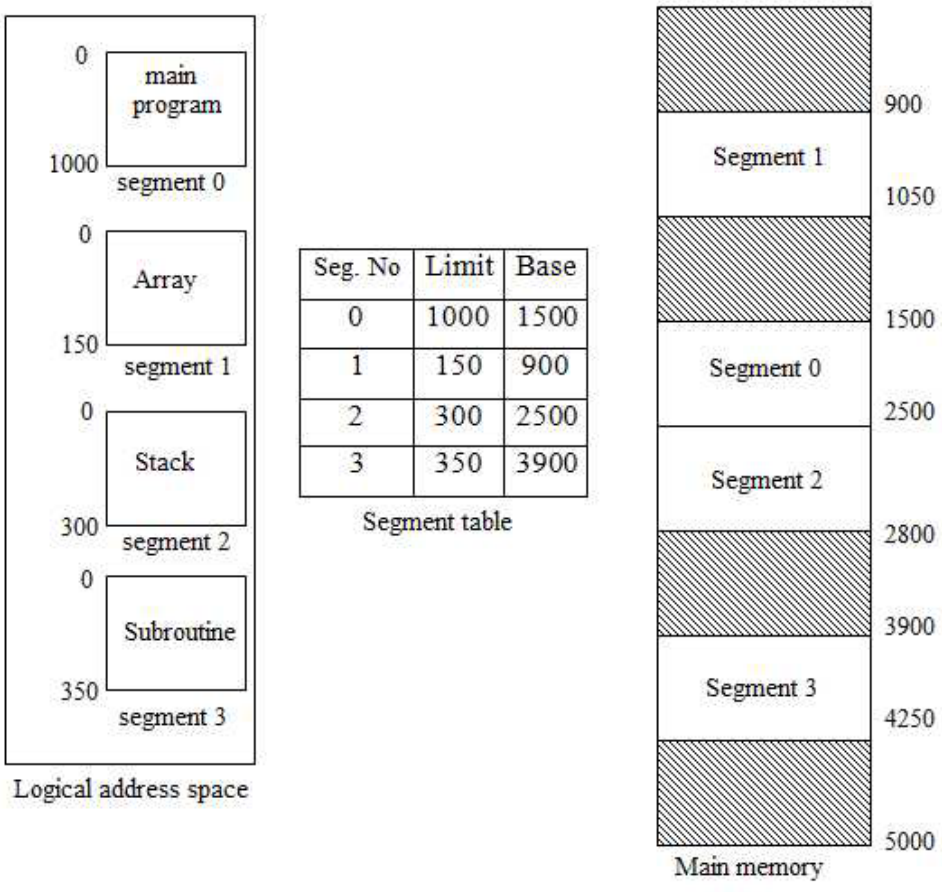


Fig 3.22 Example of segmentation

Advantages:

- . Segmentation supports virtual memory.
- It eliminates fragmentation by moving segments around; fragmented memory space can be combined in to a single free area.
- Allow dynamically growing segments
- Allows Shared segments
- Dynamic linking and loading of segments

3.1.5.6 Segmentation with paging

In this scheme segmentation is combined with paging. The process is divided into segments, then each segment is divided into pages and each segment is maintained by page table. The logical address is divided into three parts <s,p,d>. one is the segment number(s), second is the page number(p), and third is offset or displacement(d). The basic hardware for this scheme is shown in following figure. For example consider the following figure. The logical address space is divided into 3 segments numbered from 0 to 2. Each segment maintains a page table. The mapping between the page and frame is done by page table. In our example frame number 8 shows the address (1,3), 1 is a segment number and 3 is the page number. This scheme is used to avoid the fragmentation.

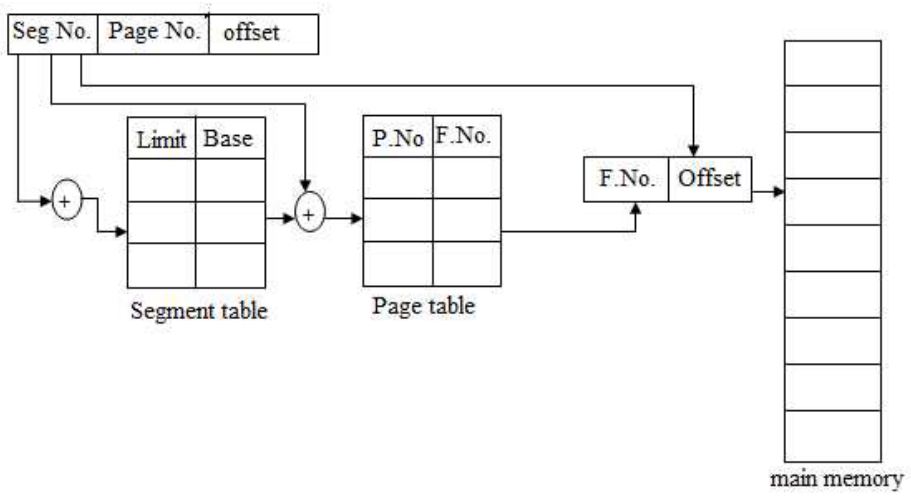


Fig 3.23 Paged segmentation memory management scheme.

3.1.6 Protection

The word protection means security from unauthorized references from main memory. To provide protection the operating system follows different type of mechanisms. In paging memory management scheme, the operating system provides protection bits that are associated with each frame. Generally these bits are kept in page table. Sometimes, these bits are said to be valid/ invalid bits. In the image below, the page map consists of a specified field that is valid/ not valid bit. If the bit shows valid, that particular page is loaded to the main memory. If on the other hand, the bit is invalid, the page is not in the processes' logical address space. Illegal addresses are trapped by using the valid/ invalid bit. The operating system sets this bit for each page to allow or disallow access to that page.

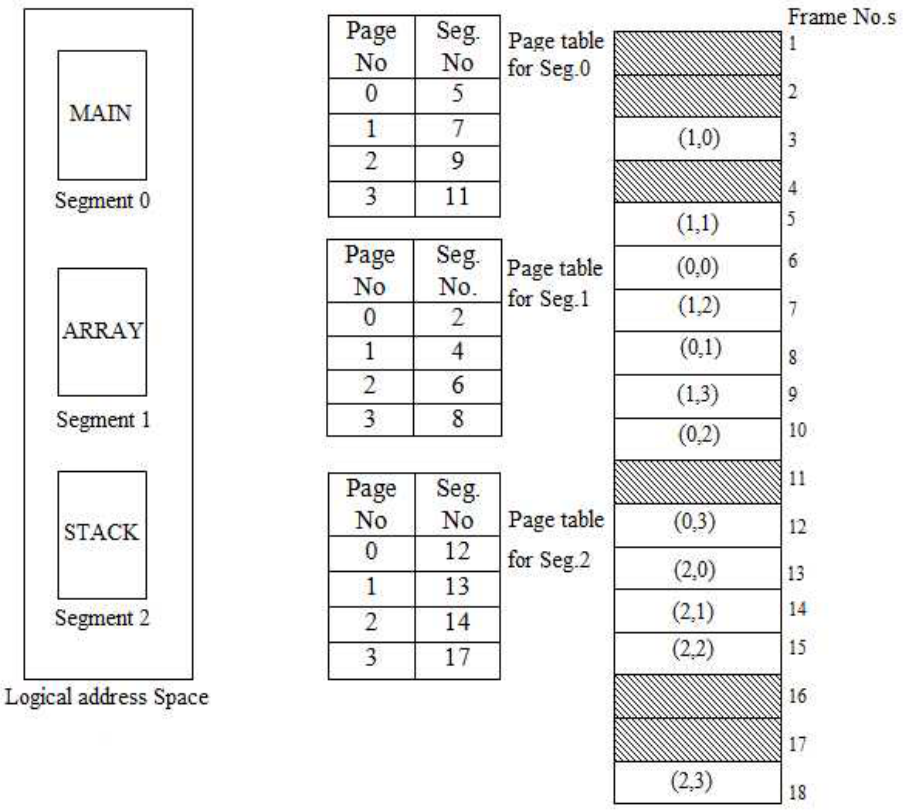


Fig 3.24 Paged Segmentation

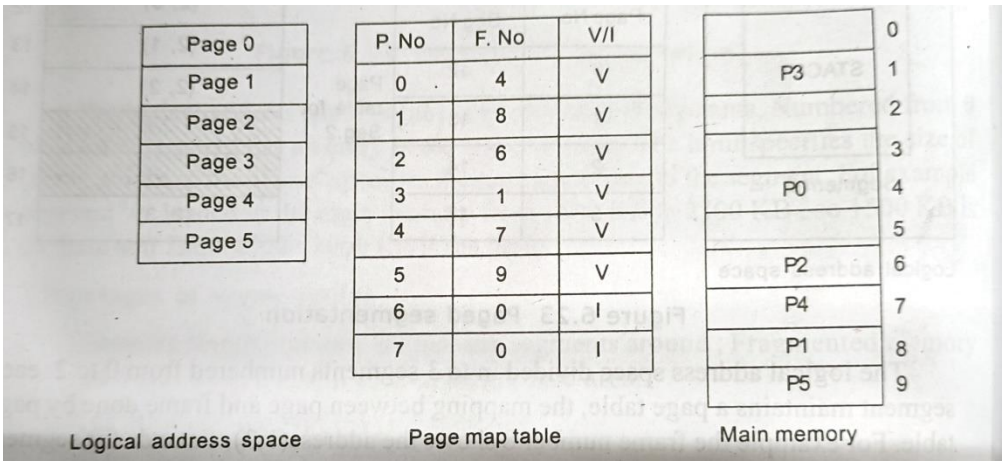


Fig 3.25 Protection in paging

Questions:

1. What is swapping?
2. What are the requirements of memory management system?
3. Discuss Dynamic Linking and loading.
4. Write briefly about the different memory allocation methods.
5. Explain Multiple partition MMS
6. What is protection in MMS?
7. Explain Segmentation and Segmentation with paging.

UNIT IV

4.1 I/O MANAGEMENT

The two main jobs of a computer system are I/O and processing. The role of the OS is to control and manage the I/O operations and I/O devices. Though there are many different devices for the input and output functions, they can be grouped in these 3 categories:

- Human Readable – They establish the communication between the computer and the user. E.g. keyboard, mouse, printer, display terminal, etc.
- Machine Readable – Communicate between electronic devices. E.g. disk drive, tape drive, sensors, controllers, etc.
- Communication – Communicate with remote devices. E.g. digital line drivers, modems.

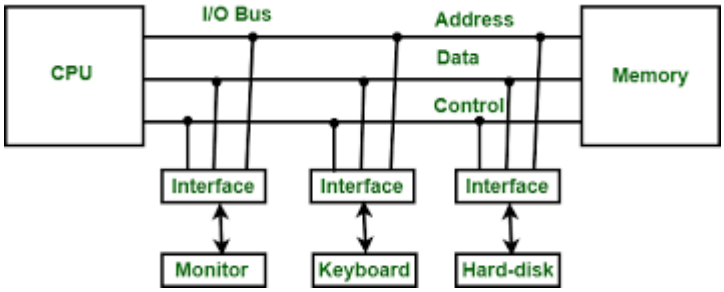


Fig 4.1 I/O device setup with CPU and Memory

The speed and function of the I/O devices vary widely and there are several different methods to handle them. To be more efficient is important to understand the aspects/ properties in which they differ.

1. **Data Rate –** The data rate is the measure of number of bits that the device can transmit and receive. It is usually measured in kB/s (kBps) or MB/s (MBps) – kiloBytes per second or MegaBytes per second. Of all the devices, the one with the least transmission speed is the keyboard at 0.01kBps and the one at the highest end is the graphics display with 30,000kBps or 30 MBps.

2. **Application –** The function of the I/O device depends on its application or usage. The same disk can be used normally or as temporary storage (spooling) or for demand paging (swap disk).

3. **Complexity of control –** A mouse or keyboard has simple control interfaces, while disks need more complex interfacing.

4. **Unit of Transfer –** The data transferred might be in streams in bytes or as characters (terminals) or in large blocks (disks).

5. **Data Representation –** Different data encoding schemes are used by I/O devices, including character code and parity conventions.

6. **Error Conditions** – The nature of errors, how to report them, consequences, actions to be taken/ responses vary largely.

Organization of I/O function

- **Programmed I/O**

Key features:

- I/O devices do not have a direct connection/ access to the memory.
- Requires the execution of a program to transfer data from I/O device to memory – hence programmed I/O
- The CPU stays in program loop till the device is ready for data transfer – consumes CPU time
- Preferred method for small low speed computers
- The CPU sends a 'Read" command to I/O devices, and wait in the program loop until it receives a response from I/O device. The I/O responds with ready/ not ready status information. The CPU checks this, if the device ready, it proceeds to process the data, if not ready, the CPU waits another cycle for the ready status of the I/O device.

Drawback – The CPU time is wasted in waiting for the ready status if I/O device.

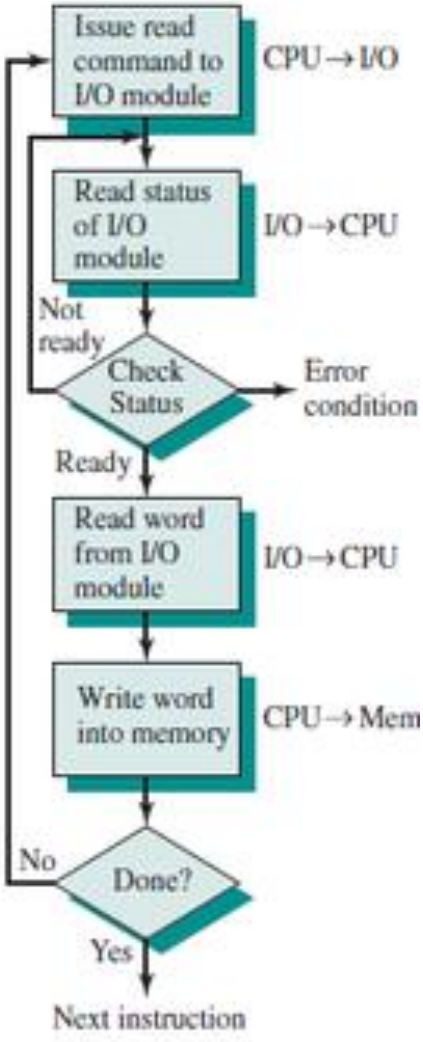


Fig 4.2 Flow chart for Programmed I/O

- **Interrupt Driven I/O**

Key features:

- The device raises a signal to the CPU whenever it is ready. This is interrupt initiated or interrupt driven I/O.
- When the interrupt signal is given to the CPU from the I/O device, the CPU checks if the device is ready, and when so, it completes the data transfer to/ from the memory from/ to I/O device.

Drawback – The CPU is on the path at the time of data transfer.

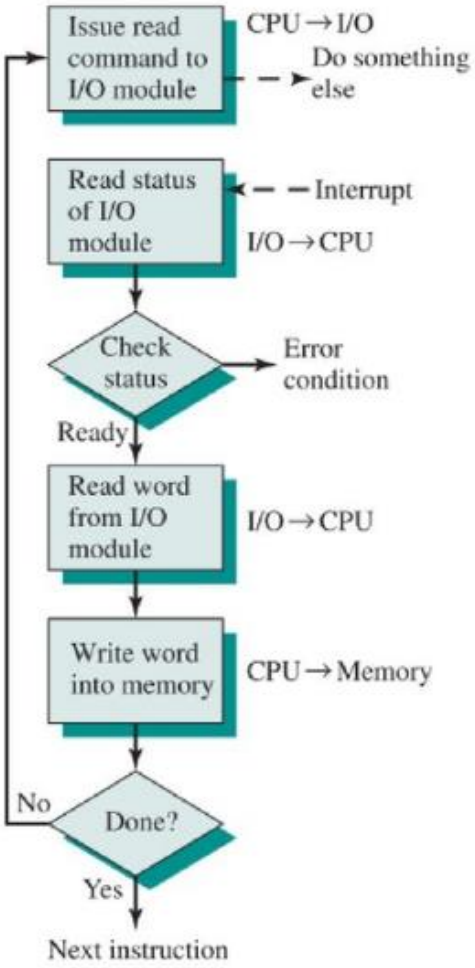


Fig 4.3 Flow chart for Interrupt initiated I/O

- **Direct Memory Access (DMA)**

Key features:

- Instead of the CPU managing the memory, the DMA controller manages it directly. It makes the transfer in less time.
- The CPU is free to manage the other tasks.
- DMA manages the buses between the I/O device and memory.
- The DMA gets information from the CPU –Address (read/write), the number of words in the block and control to start DMA transfer.

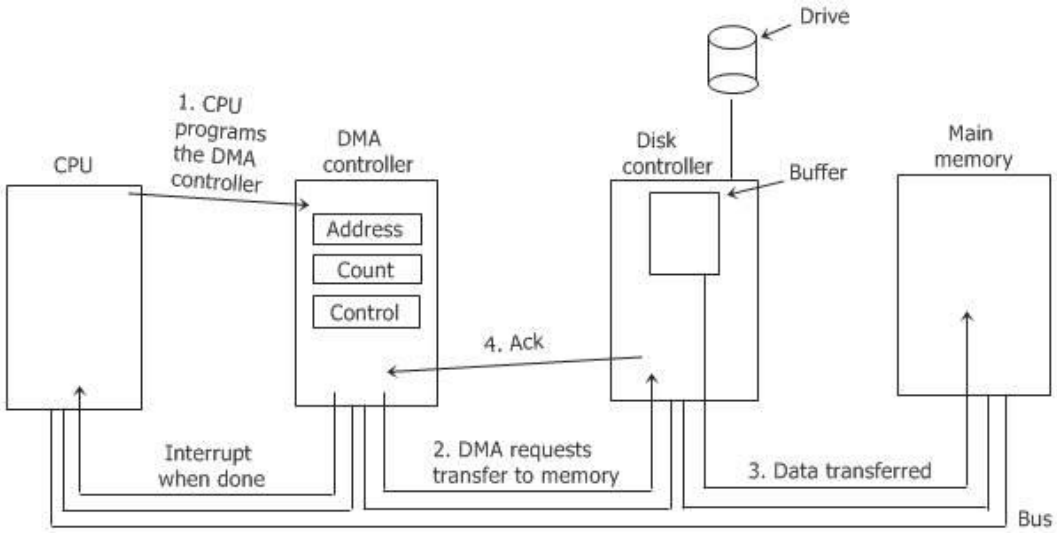


Fig 4.4 Flow chart for DMA transfer

4.1.1 I/O HARDWARE

A device communicates with a computer system by sending signals over a cable or even through air. For a device to communicate with the machine, a connection point referred to as a port is used, e.g. serial port, USB port etc. If devices share a common set of wires, the connection is called a bus. A bus is a set of wires and a rigidly defined protocol that specifies a set of messages that can be sent on the wires. Buses are used widely in computer architecture and vary in their signalling methods, speed, throughput, and connection methods and thus controllers are used. A controller is a collection of electronics that can operate a port, a bus or a device. They control signals on the wires of a port or a bus. For example, a serial-port controller is a simple device controller that control signals on the wires of a serial port.

The processor accomplishes an I/O transfer through issuing commands and data to a controller. A controller has one or more registers for data and control signals. The processor communicates with the controller by reading and writing bit patterns in these registers. I/O instructions issued by the processor to the controllers, triggers bus lines to select the proper device and to move bits into or out of a device register. Alternatively, memory-mapped I/O can be supported where device control registers are mapped into the address space of the processor. The CPU then executes I/O requests by reading and writing device-control registers at their mapped locations in physical memory.

An I/O port typically consists of four registers, called the status, control, data-in, and data-out registers.

- The data-in register is read by the host to get input.
- The data-out register is written by the host to send output.
- The status register contains bits that can be read by the host. These bits indicate states, such as whether the current command has completed, whether a byte is available to be read from the data-in register, and whether a device error has occurred.
- The control register can be written by the host to start a command or to change the mode of a device. For instance, changing into different speeds supported or word length from smaller bits to higher bits.

~~A device communicates with a computer system by sending signals over a cable or even through air. For a device to communicate with the machine, a connection point referred to as a port is used, e.g. serial port, USB port etc. If devices share a common set of wires, the connection is called a bus. A bus is a set of wires and a rigidly defined protocol that specifies a set of messages that can be sent on the wires. Buses are used widely in computer architecture and vary in their signalling methods, speed, throughput, and connection methods and thus controllers are used. A controller is a collection of electronics that can operate a port, a bus or a device. They control signals on the wires of a port or a bus. For example, a serial port controller is a simple device controller that control signals on the wires of a serial port.~~

A common set of wires connecting multiple devices is termed a ***bus.***

- Buses include rigid protocols for the types of messages that can be sent across the bus and the procedures for resolving contention issues.
- Figure below illustrates three of the four bus types commonly found in a modern PC:

1. The ***PCI bus*** connects high-speed high-bandwidth devices to the memory subsystem (and the CPU.)
2. The ***expansion bus*** connects slower low-bandwidth devices, which typically deliver data one character at a time (with buffering.)
3. The ***SCSI bus*** connects a number of SCSI devices to a common SCSI controller.
4. A ***daisy-chain bus,*** (not shown) is when a string of devices is connected to each other like beads on a chain, and only one of the devices is directly connected to the host.

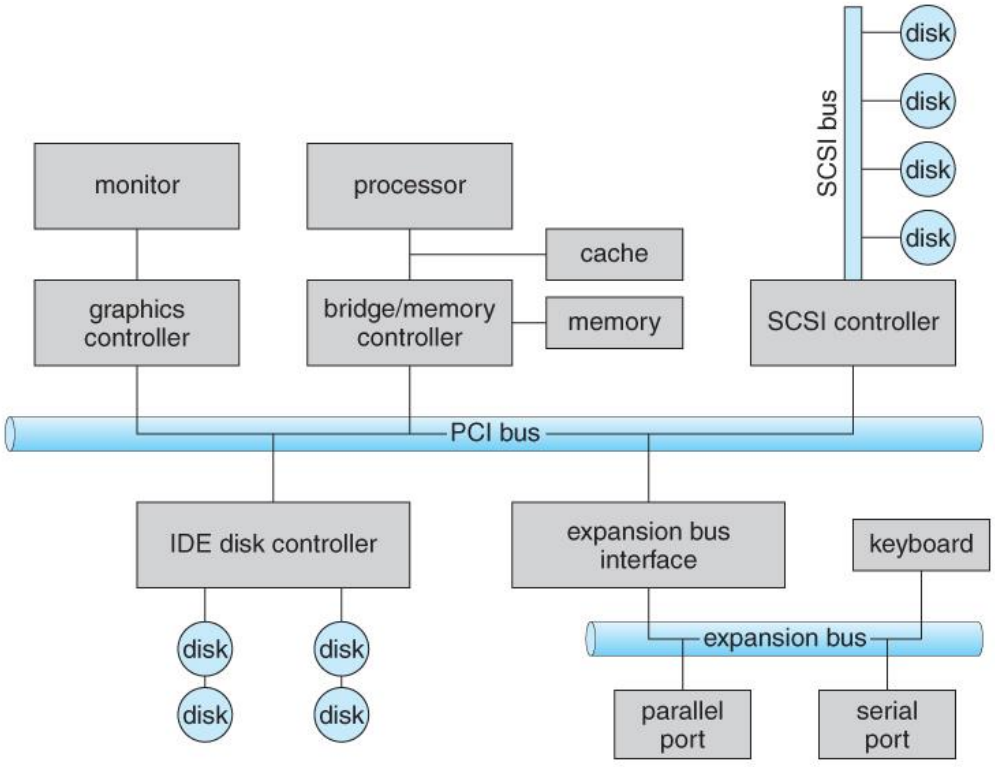


Fig 4.5 PC Bus Architecture

The processor accomplishes an I/O transfer through issuing commands and data to a controller. A controller has one or more registers for data and control signals. The processor communicates with the controller by reading and writing bit patterns in these registers. I/O instructions issued by the processor to the controllers, triggers bus lines to select the proper device and to move bits into or out of a device register. Alternatively, memory-mapped I/O can be supported where device control registers are mapped into the address space of the processor. The CPU then executes I/O requests by reading and writing device-control registers at their mapped locations in physical memory. An I/O port typically consists of four registers, called the status, control, data-in, and data-out registers.

- The data-in register is read by the host to get input.
- The data-out register is written by the host to send output.
- The status register contains bits that can be read by the host. These bits indicate states, such as whether the current command has completed, whether a byte is available to be read from the data-in register, and whether a device error has occurred.
- The control register can be written by the host to start a command or to change the mode of a device. For instance, changing into different speeds supported or word length from smaller bits to higher bits

4.1.2 I/O BUFFERING

Suppose that a user process wishes to read blocks of data from a disk one at a time, with each block having a length of 512 bytes. The data are to be read into a data area within the address space of the user process at virtual location 1000 to 1511.The simplest way would be to execute an I/O command (something like Read_Block [1000, disk]) to the disk unit and then wait for the data to become available. The waiting could either be busy waiting (continuously test the device status) or, more practically, process suspension on an interrupt.

There are two problems with this approach. First, the program is hung up waiting for the relatively slow I/O to complete. The second problem is that this approach to I/O interferes with swapping decisions by the operating system. Virtual locations 1000 to 1511 must remain in main memory during the course of the block transfer. Otherwise, some of the data may be lost. If paging is being used, at least the page containing the target locations must be locked into main memory. Thus, although portions of the process may be paged out to disk, it is impossible to swap the process out completely, even if this is desired by the operating system.

To avoid these overheads and inefficiencies, it is sometimes convenient to perform input transfers in advance of requests being made and to perform output transfers sometime after the request is made. This technique is known as buffering. In discussing the various approaches to buffering, it is sometimes important to make a distinction between two types of I/O devices: block oriented and stream oriented.

Single Buffer

The simplest type of support that the operating system can provide is single buffering. When a user process issues I/O request, the operating system assigns a buffer in the system

portion of main memory to the operation. For block-oriented devices, the single buffering scheme can be described as follows:

Input transfers are made to the system buffer. When the transfer is complete, the process moves the block into user space and immediately requests another block. This is called reading ahead, or anticipated input; it is done in the expectation that the block will eventually be needed. For many types of computation, this is a reasonable assumption most of the time because data are usually accessed sequentially. Only at the end of a sequence of processing will a block be read in unnecessarily.

Double Buffer

An improvement over single buffering can be had by assigning two system buffers to the operation. A process now transfers data to (or from) one buffer while the operating system empties (or fills) the other. This technique is known as double buffering or buffer swapping.

Circular Buffer

A double-buffer scheme should smooth out the flow of data between an I/O device and a process. If the performance of a particular process is the focus of our concern, then we would like for the I/O operation to be able to keep up with the process. Double buffering may be inadequate if the process performs rapid bursts of I/O. In this case, the problem can often be alleviated by using more than two buffers.

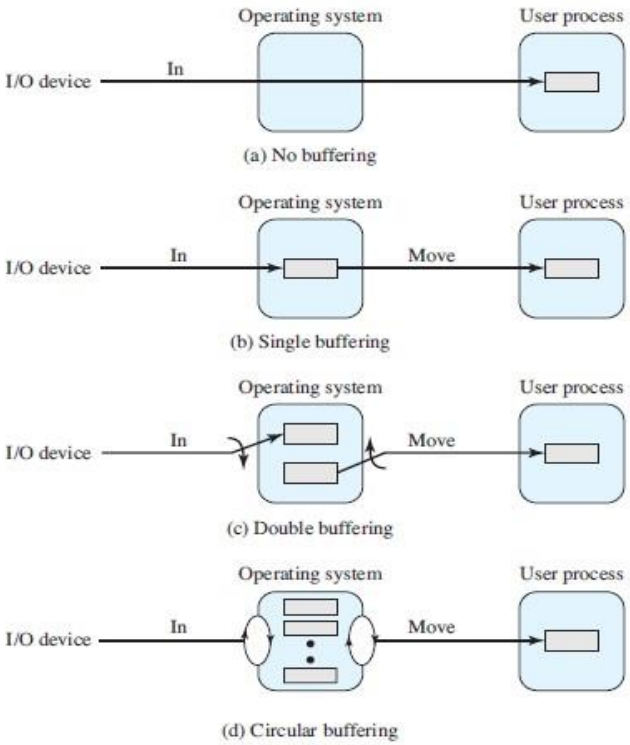


Fig 4.6 I/O Buffering Schemes (input)

4.1.3 DISK I/O

In modern computers, most of the secondary storage is in the form of magnetic disks. Hence, knowing the structure of a magnetic disk is necessary to understand how the data in the disk is accessed by the computer.

A magnetic disk contains several platters. Common platter diameter is 1.8 to 5.25inches. Each platter is divided into circular shaped tracks. The length of the tracks near the centre is less than the length of the tracks farther from the centre. Each track is further divided into sectors, as shown in the figure. Tracks of the same distance from centre form a cylinder. A read-write head is used to read data from a sector of the magnetic disk.

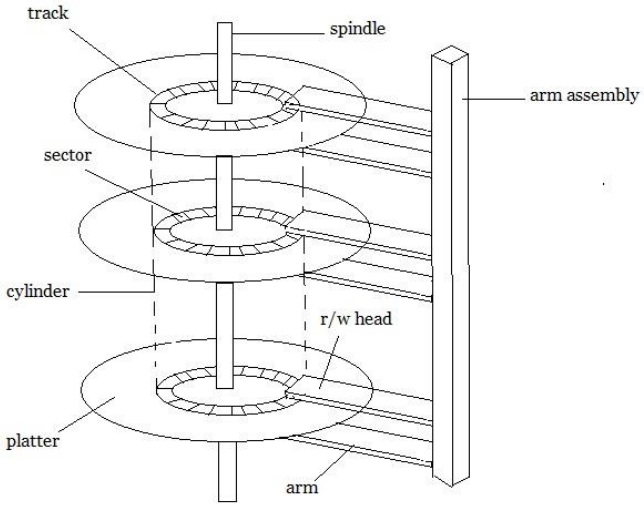


Fig 4.7 Disk Structure

Disk Performance Parameters

Seek Time

Seek time is the time required to move the disk arm to the required track. It turns out that this is a difficult quantity to pin down. The seek time consists of two key components: the initial startup time and the time taken to traverse the tracks that have to be crossed once the access arm is up to speed. Unfortunately, the traversal time is not a linear function of the number of tracks but includes a settling time (time after positioning the head over the target track until track identification is confirmed).Much improvement comes from smaller and lighter disk components. Some years ago, a typical disk was 14 inches (36 cm) in diameter, whereas the most common size today is 3.5 inches (8.9 cm), reducing the distance that the arm has to travel. A typical average seek time on contemporary hard disks is under 10 ms.

The linear formula for seek time is

$$T_s = mn + s$$

T_s – Estimated seek time
n – no. of tracks traversed
m – Constant that depends on disk
s – Start up time

Rotational Delay

Rotational delay is the time required for the addressed area of the disk to rotate into a position where it is accessible by the read/write head. Disks, other than floppy disks, rotate at speeds ranging from 3600 rpm (for handheld devices such as digital cameras) up to, as of

this writing, 15,000 rpm; at this latter speed, there is one revolution per 4 ms. Thus, on the average, the rotational delay will be 2 ms. Floppy disks typically rotate at between 300 and 600 rpm. Thus the average delay will be between 100 and 50 ms.

Transfer Time

The transfer time to or from the disk depends on the rotation speed of the disk in the following fashion:

$$T = b/rN$$

T = transfer time
b = number of bytes to be transferred
N = number of bytes on a track
r = rotation speed, in revolutions per second

The total average access time

$T_a = T_s + 1/2R + B/RN$

Disk Scheduling Algorithms

Consider the typical situation in a multiprogramming environment, in which the operating system maintains a queue of requests for each I/O device. So, for a single disk, there will be a number of I/O requests (reads and writes) from various processes in the queue. If we selected items from the queue in random order, then we can expect that the tracks to be visited will occur randomly, giving poor performance. This random scheduling is useful as a benchmark against which to evaluate other techniques.

First-In-First-Out

The simplest form of scheduling is first-in-first-out (FIFO) scheduling, which processes items from the queue in sequential order. This strategy has the advantage of being fair, because every request is honored and the requests are honored in the order received. With FIFO, if there are only a few processes that require access and if many of the requests are to clustered file sectors, then we can hope for good performance. However, this technique will often approximate random scheduling in performance, if there are many processes competing for the disk. Thus, it may be profitable to consider a more sophisticated scheduling policy.

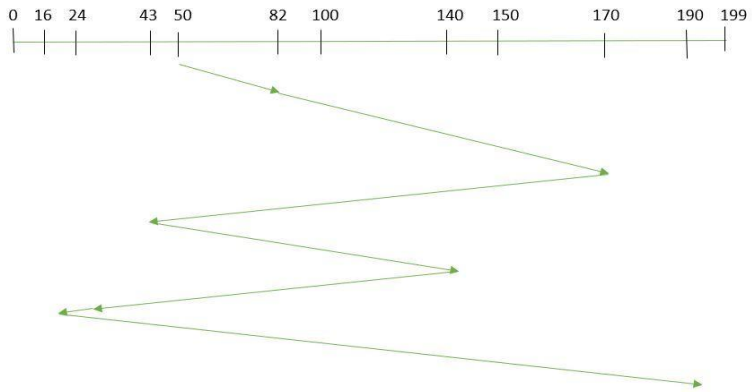
Example: Suppose the order of request is- (82,170,43,140,24,16,190) And current position of Read/Write head is : 50

So, total seek time:

= (82 - 50) + (170 - 82) + (170 - 43) + (140 - 43) + (140 - 24) + (24 - 16) + (190 - 16)
= 642

Advantages:

- Every request gets a fair chance
- No indefinite postponement



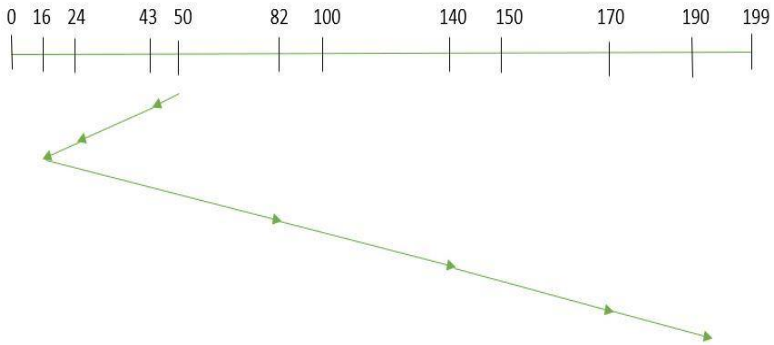
Disadvantages:

- Does not try to optimize seek time
- May not provide the best possible service

SSTF

In SSTF (Shortest Seek Time First), requests having shortest seek time are executed first. So, the seek time of every request is calculated in advance in the queue and then they are scheduled according to their calculated seek time. As a result, the request near the disk arm will get executed first. SSTF is certainly an improvement over FCFS as it decreases the average response time and increases the throughput of system. Let us understand this with the help of an example.

Example: Suppose the order of request is- (82,170,43,140,24,16,190)
And current position of Read/Write head is : 50



So, total seek time:
= (50 - 43) + (43 - 24) + (24 - 16) + (82 - 16) + (140 - 82) + (170 - 140) + (190 - 170)
=208

Advantages:

- Average Response Time decreases
- Throughput increases

Disadvantages:

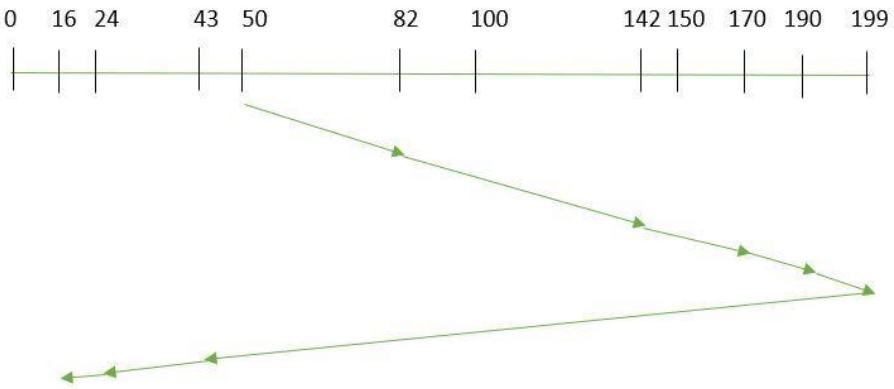
- Overhead to calculate seek time in advance

- Can cause Starvation for a request if it has higher seek time as compared to incoming requests
- High variance of response time as SSTF favours only some requests

SCAN

In SCAN algorithm the disk arm moves into a particular direction and services the requests coming in its path and after reaching the end of disk, it reverses its direction and again services the request arriving in its path. So, this algorithm works as an elevator and hence also known as elevator algorithm. As a result, the requests at the midrange are serviced more and those arriving behind the disk arm will have to wait.

Example: Suppose the requests to be addressed are-82,170,43,140,24,16,190. And the Read/Write arm is at 50, and it is also given that the disk arm should move "towards the larger value".



Therefore, the seek time is calculated as:

= (199 - 50) + (199 - 16)

= 332

Advantages:

- High throughput
- Low variance of response time
- Average response time

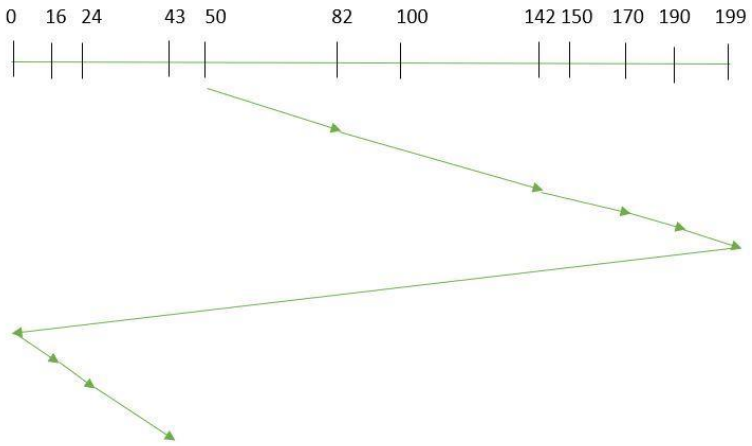
Disadvantages:

- Long waiting time for requests for locations just visited by disk arm

CSCAN

In SCAN algorithm, the disk arm again scans the path that has been scanned, after reversing its direction. So, it may be possible that too many requests are waiting at the other end or there may be zero or few requests pending at the scanned area. These situations are avoided in *CSCAN* algorithm in which the disk arm instead of reversing its direction goes to the other end of the disk and starts servicing the requests from there. So, the disk arm moves in a circular fashion and this algorithm is also similar to SCAN algorithm and hence it is known as C-SCAN (Circular SCAN).

Example: Suppose the requests to be addressed are-82,170,43,140,24,16,190. And the Read/Write arm is at 50, and it is also given that the disk arm should move “towards the larger value”.



Seek time is calculated as:
= (199 - 50) + (199 - 0) + (43 - 0)
= 391

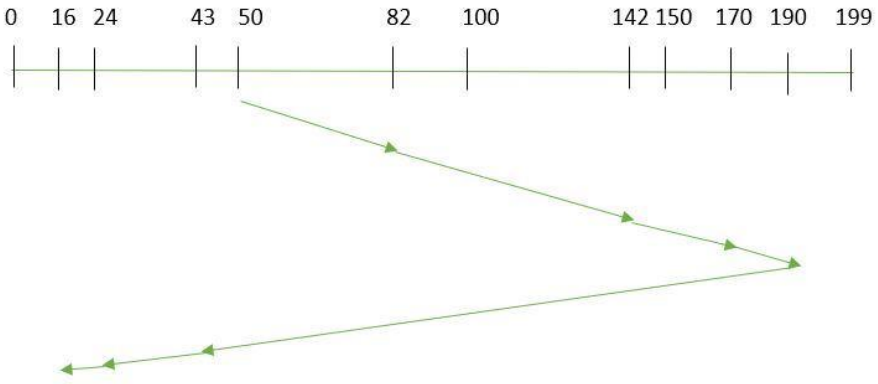
Advantages:

- Provides more uniform wait time compared to SCAN

LOOK

It is similar to the SCAN disk scheduling algorithm except for the difference that the disk arm in spite of going to the end of the disk goes only to the last request to be serviced in front of the head and then reverses its direction from there only. Thus it prevents the extra delay which occurred due to unnecessary traversal to the end of the disk.

Example: Suppose the requests to be addressed are-82,170,43,140,24,16,190. And the Read/Write arm is at 50, and it is also given that the disk arm should move “towards the larger value”.



So, the seek time is calculated as:
= (190 - 50) + (190 - 16)
= 314

4.1.4 RAID

RAID, or “Redundant Arrays of Independent Disks” is a technique which makes use of a combination of multiple disks instead of using a single disk for increased performance, data redundancy or both. The term was coined by David Patterson, Garth A. Gibson, and Randy Katz at the University of California, Berkeley in 1987.

Why data redundancy?

Data redundancy, although taking up extra space, adds to disk reliability. This means, in case of disk failure, if the same data is also backed up onto another disk, we can retrieve the data and go on with the operation. On the other hand, if the data is spread across just multiple disks without the RAID technique, the loss of a single disk can affect the entire data.

Key evaluation points for a RAID System

- Reliability: How many disk faults can the system tolerate?
- Availability: What fraction of the total session time is a system in uptime mode, i.e. how available is the system for actual use?
- Performance: How good is the response time? How high is the throughput (rate of processing work)? Note that performance contains a lot of parameters and not just the two.
- Capacity: Given a set of N disks each with B blocks, how much useful capacity is available to the user?

RAID is very transparent to the underlying system. This means, to the host system, it appears as a single big disk presenting itself as a linear array of blocks. This allows older technologies to be replaced by RAID without making too many changes in the existing code.

Different RAID levels

RAID-0 (Stripping)

- Blocks are “stripped” across disks.

Disk 0	Disk 1	Disk 2	Disk 3
0	1	2	3
4	5	6	7
8	9	10	11
12	13	14	15

- In the figure, blocks “0,1,2,3” form a stripe.
- Instead of placing just one block into a disk at a time, we can work with two (or more) blocks placed into a disk before moving on to the next one.

Disk 0	Disk 1	Disk 2	Disk 3
0	3	4	6
1	3	5	7
8	10	12	14
9	11	13	15

Evaluation:

- Reliability: 0
  There is no duplication of data. Hence, a block once lost cannot be recovered.
- Capacity: N*B
  The entire space is being used to store data. Since there is no duplication, N disks each having B blocks are fully utilized.

RAID-1 (Mirroring)

- More than one copy of each block is stored in a separate disk. Thus, every block has two (or more) copies, lying on different disks.

Disk 0	Disk 1	Disk 2	Disk 3
0	0	1	1
2	2	3	3
4	4	5	5
6	6	7	7

- The above figure shows a RAID-1 system with mirroring level 2.
- RAID 0 was unable to tolerate any disk failure. But RAID 1 is capable of reliability.

Evaluation:

Assume a RAID system with mirroring level 2.

- Reliability: 1 to N/2
  1 disk failure can be handled for certain, because blocks of that disk would have duplicates on some other disk. If we are lucky enough and disks 0 and 2 fail, then again this can be handled as the blocks of these disks have duplicates on disks 1 and 3. So, in the best case, N/2 disk failures can be handled.
- Capacity: N*B/2
  Only half the space is being used to store data. The other half is just a mirror to the already stored data.

RAID-4 (Block-Level Stripping with Dedicated Parity)

- Instead of duplicating data, this adopts a parity-based approach.

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
4	5	6	7	P1
8	9	10	11	P2
12	13	14	15	P3

- In the figure, we can observe one column (disk) dedicated to parity.
- Parity is calculated using a simple XOR function. If the data bits are 0,0,0,1 the parity bit is XOR(0,0,0,1) = 1. If the data bits are 0,1,1,0 the parity bit is XOR(0,1,1,0) = 0. A simple approach is that even number of ones results in parity 0, and an odd number of ones results in parity 1.

C1	C2	C3	C4	Parity
0	0	0	1	1
0	1	1	0	0

- Assume that in the above figure, C3 is lost due to some disk failure. Then, we can precompute the data bit stored in C3 by looking at the values of all the other columns and the parity bit. This allows us to recover lost data.

Evaluation:

- Reliability: 1
  RAID-4 allows recovery of at most 1 disk failure (because of the way parity works). If more than one disk fails, there is no way to recover the data.
- Capacity: (N-1)*B
  One disk in the system is reserved for storing the parity. Hence, (N-1) disks are made available for data storage, each disk having B blocks.

RAID-5 (Block-Level Stripping with Distributed Parity)

- This is a slight modification of the RAID-4 system where the only difference is that the parity rotates among the drives.

Disk 0	Disk 1	Disk 2	Disk 3	Disk 4
0	1	2	3	P0
5	6	7	P1	4
10	11	P2	8	9
15	P3	12	13	14
P4	16	17	18	19

- In the figure, we can notice how the parity bit “rotates”.

- This was introduced to make the random write performance better.

Evaluation:

- Reliability: 1
  RAID-5 allows recovery of at most 1 disk failure (because of the way parity works). If more than one disk fails, there is no way to recover the data. This is identical to RAID-4.

- Capacity: (N-1)*B
  Overall, space equivalent to one disk is utilized in storing the parity. Hence, (N-1) disks are made available for data storage, each disk having B blocks.

4.1.5 DISK CACHE

The term cache memory is usually used to apply to a memory that is smaller and faster than main memory and that is interposed between main memory and the processor. Such a cache memory reduces average memory access time by exploiting the principle of locality. The same principle can be applied to disk memory. Specifically, a disk cache is a buffer in main memory for disk sectors. The cache contains a copy of some of the sectors on the disk. When an I/O request is made for a particular sector, a check is made to determine if the sector is in the disk cache. If so, the request is satisfied via the cache. If not, the requested sector is read into the disk cache from the disk. Because of the phenomenon of locality of reference, when a block of data is fetched into the cache to satisfy a single I/O request, it is likely that there will be future references to that same block. Several design issues are of interest.

First, when an I/O request is satisfied from the disk cache, the data in the disk cache must be delivered to the requesting process.This can be done either by transferring the block of data within main memory from the disk cache to memory assigned to the user process, or simply by using a shared memory capability and passing a pointer to the appropriate slot in the disk cache.The latter approach saves the time of a memory-to-memory transfer and also allows shared access by other processes using the readers/writers model.

A second design issue has to do with the replacement strategy. When a new sector is brought into the disk cache, one of the existing blocks must be replaced. A number of algorithms have been tried. The most commonly used algorithm is least recently used (LRU): Replace that block that has been in the cache longest with no reference to it. Logically, the cache consists of a stack of blocks, with the most recently referenced block on the top of the stack. When a block in the cache is referenced, it is moved from its existing position on the stack to the top of the stack. When a block is brought in from secondary memory, remove the block that is on the bottom of the stack and push the incoming block onto the top of the stack. Naturally, it is not necessary actually to move these blocks around in main memory; a stack of pointers can be associated with the cache.

Another possibility is least frequently used (LFU): Replace that block in the set that has experienced the fewest references. LFU could be implemented by associating a counter with each block. When a block is brought in, it is assigned a count of 1; with each reference to the block, its count is incremented by 1.When replacement is required, the block with the

smallest count is selected. Intuitively, it might seem that LFU is more appropriate than LRU because LFU makes use of more pertinent information about each block in the selection process.

A simple LFU algorithm has the following problem. It may be that certain blocks are referenced relatively infrequently overall, but when they are referenced, there are short intervals of repeated references due to locality, thus building up high reference counts. After such an interval is over, the reference count may be misleading and not reflect the probability that the block will soon be referenced again. Thus, the effect of locality may actually cause the LFU algorithm to make poor replacement choices.

To overcome this difficulty with LFU, a technique known as frequency-based replacement is implemented. For clarity, let us first consider a simplified version, illustrated in Figure 4.2a. The blocks are logically organized in a stack, as with the LRU algorithm. A certain portion of the top part of the stack is designated the new section. When there is a cache hit, the referenced block is moved to the top of the stack. If the block was already in the new section, its reference count is not incremented; otherwise it is incremented by 1. Given a sufficiently large new section, this would result in the reference counts for blocks that are repeatedly re-referenced within a short interval remaining unchanged. On a miss, the block with the smallest reference count that is not in the new section is chosen for replacement; the least recently used such block is chosen in the event of a tie.

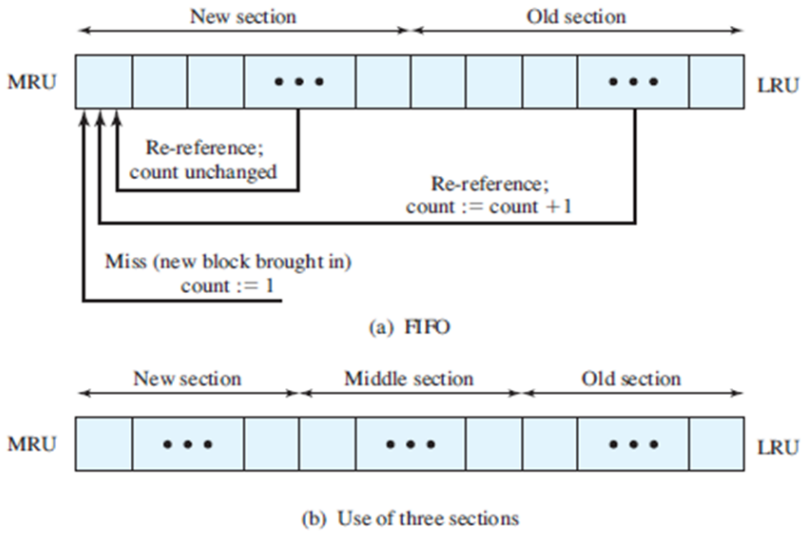


Fig 4.8 Frequency-Based Replacement

4.2 FILE MANAGEMENT

The file system is the most visible aspect of an operating system. It 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 Attributes

A file is named, for the convenience of its human users, and is referred to by its name. A name is usually a string of characters, such as *'exm.c'*. Some systems differentiate between

uppercase and lowercase characters in names, whereas other systems do not. When a file is named, it becomes independent of the process, the user, and even the system that created it. The information about all files is kept in the directory structure, which also resides on secondary storage. A file has certain other attributes, which vary from one operating system to another but typically consist of these:

- Name: The symbolic file name is the only information kept in human readable form.
- Identifier: This unique tag, usually a number, identifies the file within the file system; it is the non-human-readable name for the file.
- Type: This information is needed for those systems that support different types of files.
- Location: This information is a pointer to a device and to the location of the file on that 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, last modification, and last use. These data can be useful for protection, security, and usage monitoring.

Types of files:

A common technique for implementing file types is to include the type as part of the file name. The name is split into two parts—a name and an extension, usually separated by a period character. In this way, the user and the operating system can tell from the name alone what the type of a file is. For example, in MSDOS, a name can consist of up to eight characters followed by a period and terminated by an extension of up to three characters. The system uses the extension to indicate the type of the file and the type of operations that can be done on that file. Only a file with a .com, .exe, or .bat extension can be *executed*, for instance. The *.com* and *.exe* files are two forms of binary executable files, whereas a *.bat* file is a batch file containing, in ASCII format, commands to the operating system. MSDOS recognizes only a few extensions, but application programs also use extensions to indicate file types in which they are interested. For example, assemblers expect source files to have an *.asm* extension, and the WordPerfect word processor expects its file to end with a *.wp* extension. Following table shows some common file types.

file type	usual extension	function
executable	exe, com, bin or none	ready-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, rtf, doc	various word-processor formats

file type	usual extension	function
library	lib, a, so, dll	libraries of routines for programmers
print or view	ps, pdf, jpg	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

Operations on Database Files:

- Retrieve-one: This command retrieves one record at a time, interactive, transaction oriented applications needs this command.
- Retrieve-next: This command retrieves the next record – the record next to the one recently retrieved. E.g. Filling forms
- Retrieve-previous: This command retrieves the previous record – the record before to the one recently retrieved.
- Retrieve-few: This command retrieves a number of records. E.g. Accessing all records that satisfy some criteria.
- Insert-one: This command inserts a new record to the existing database file.
- Delete-one: This command deletes the record in the file specified by user.
- Update-one: This command updates one or more fields and rewrites the updated record back into the files.

Operations on Programmable Files:

A file is an abstract data type. The basic operation performed on files are create file, write to a file, read a file, reposition a file, delete a file. The operating system can provide system calls to perform all these operations.

- 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, its 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 file's location. The directory entry will need to store a pointer to the current end of the file. Using this pointer the address of the next block can be computed and the information can be written. The write pointer must be updated.

- Reading a file:

To read from a file, we use a system call that specifies the name of the file and memory location where the next block of the file should be put. Again the directory is searched for the associated directory entry. And again the directory will need a pointer to the next block to be read. Once that block is read , the pointer is updated. A given process is usually only reading or writing a given file, and the current operation location is kept as a per-process current-file-position pointer. Both the read and write operations use this same pointer, saving space and reducing the system complexity.

- Repositioning within a file:

The directory is searched for the appropriate entry, and the current- file-position pointer is set to a given value (given position). Repositioning within a file need not involve any actual I/O. This file operation is also known as files seek.

- 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.

4.2.1 FILE MANAGEMENT SYSTEM

File management is one of the most visible services of an operating system. Computer can store information in several different physical forms; magnetic tape, disk, optical disk are the most common forms. Each of these devices has its own characteristics and physical organization. The operating system abstracts from the physical properties of its storage devices to define a logical storage unit, the file. Files are mapped by the operating system onto physical devices. These storage devices are usually nonvolatile, so the contents are persistent through power failures and system reboots.

A file is a named collection of related information that is recorded on secondary storage. data cannot be written to secondary storage unless they are within a file. Commonly, files represent programs (both source and object forms) and data. Data files may be numeric, alphabetic, alphanumeric, or binary. Files may be free form, such as text files, or may be formatted rigidly. In general, a file is a sequence of bits, bytes, lines, or records, the meaning of which is defined by the file's creator and user.

The information in a file is defined by its creator. Many different types of information may be stored in a file—source programs, object programs, executable programs, numeric data, text, payroll records, graphic images, sound recordings, and so on. A file has a certain defined structure, which depends on its type. A text file is a sequence of characters organized into lines (and possibly pages). A source file is a sequence of subroutines and functions, each of which is further organized as declarations followed by executable statements. An object file is a sequence of bytes organized into blocks understandable by the system's linker. An executable file is a series of code sections that the loader can bring into memory and execute.

The main functions/ objectives of the file management system are the follows:

- It provides I/O support for a variety of storage device types.

- Minimizes the chances of lost or destroyed data
- Helps OS to standardized I/O interface routines for user processes.
- It provides I/O support for multiple users in a multiuser systems environment.
- It optimizes the performance
- Storage of data

File System Architecture

- Device drivers - the lowest level software that communicates with the I/O hardware; responsible for control and completion of an I/O request through the OS.
- Basic File System - this level handles the physical I/O operations, the buffering, manages the organization of the physical layout of the device.
- Basic I/O supervisor- scheduling and optimization management.
- Logical I/O - user level access. Blocking and unblocking of logical records, tracking of basic file information (pointers, sizes, locations)
- Access Methods - interface between the applications and the system. The file types and the associated operations are provided at this level.

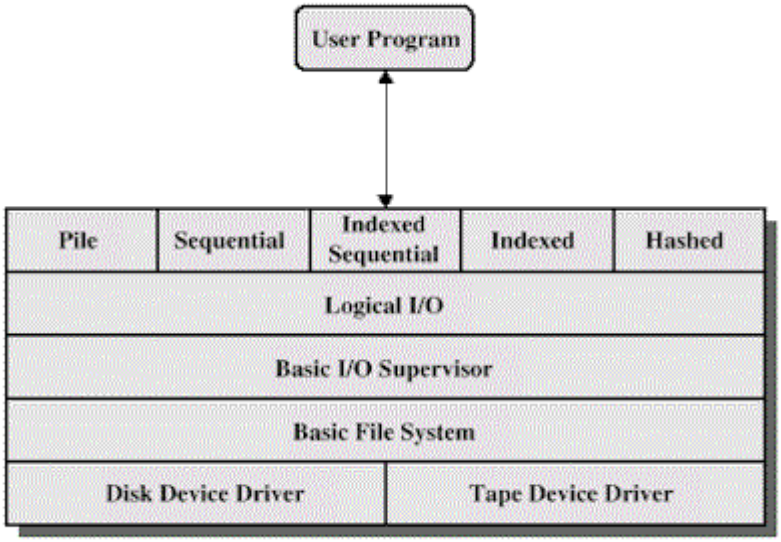


Fig 4.9 File System Architecture

4.2.2 FILE ACCESSING METHODS

Files store information. When it is used, this information must be accessed and read into computer memory. The information in the file can be accessed in several ways. Some systems provide only one access method for files. Other systems, such as those of IBM, support many access methods, and choosing the right one for a particular application is a major design problem.

Sequential access method:

The simplest access method is sequential access. Information in the file is processed in order, one record after the other. Reads and writes make up the bulk of the operations on a file. 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). Such a file can be reset to the beginning; and on some systems, a program may be able to skip forward or backward n records, for some integer n— perhaps only for n = 1. Sequential access, which is depicted in figure, is based on a tape model of a file and works as well on sequential-access devices as it does on random-access ones.

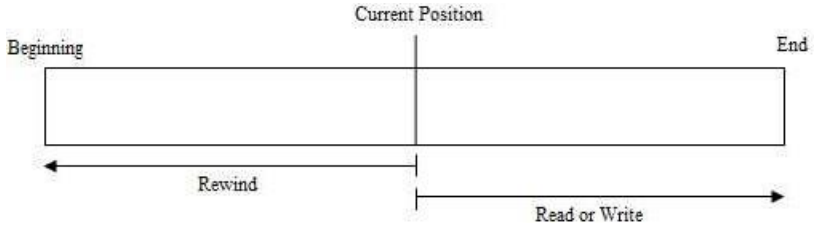


Fig 4.10 Sequential access file

Direct Access:

Another method is direct access (or relative access). . The direct- access method is based on a disk model of a file, since disks allow random access to any file block. A file is made up of fixed length logical records that allow programs to read and write records rapidly in no particular order For direct access, the file is viewed as a numbered sequence of blocks or records. A direct-access file allows blocks to be read or written in any order. Thus, we may read block 14, then read block 53, and then write block 7. There are no restrictions on the order of reading or writing for a direct- access file. Direct-access files are of great use for immediate access to large amounts of information. Databases are often of this type. 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.

The block number provided by the user to the operating system is normally a relative block number. A relative block number is an index relative to the beginning of the file. Thus, the first relative block of the file is 0, the next is 1, and so on, even though the actual absolute disk address of the block may be 14703 for the first block and 3192 for the second. The use of relative block numbers allows the operating system to decide where the file should be placed and helps to prevent the user from accessing portions of the file system that may not be part of her file. Some systems start their relative block numbers at 0; others start at 1.

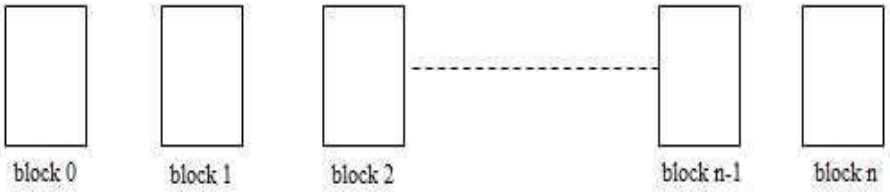


Fig 4.11 Direct access file

Indexed Sequential file:

Indexed file organization is the storage of records either sequentially or non-sequentially with an index that allows software to locate individual records. An index is a table or other data structure used to determine the location of rows in a file that satisfy some condition. Each index entry matches a key value with one or more records. An index can point to unique records (a primary key index) or potentially more than one record.

A secondary key is one field or a combination of fields for which more than one record may have the same combination of values, which is also called a non-unique key.

IBM‘s indexed sequential access method uses a small master index which points to disk blocks of a secondary index. The secondary index block point to the actual file blocks. Then to find a particular item, the binary search is first made of the master index, which provides the block number of the secondary index. This block is read in, and again a binary search is used to find the block containing the desired record. Finally this block is searched sequentially. In this way, any record can be located from its key by at most two direct access reads.

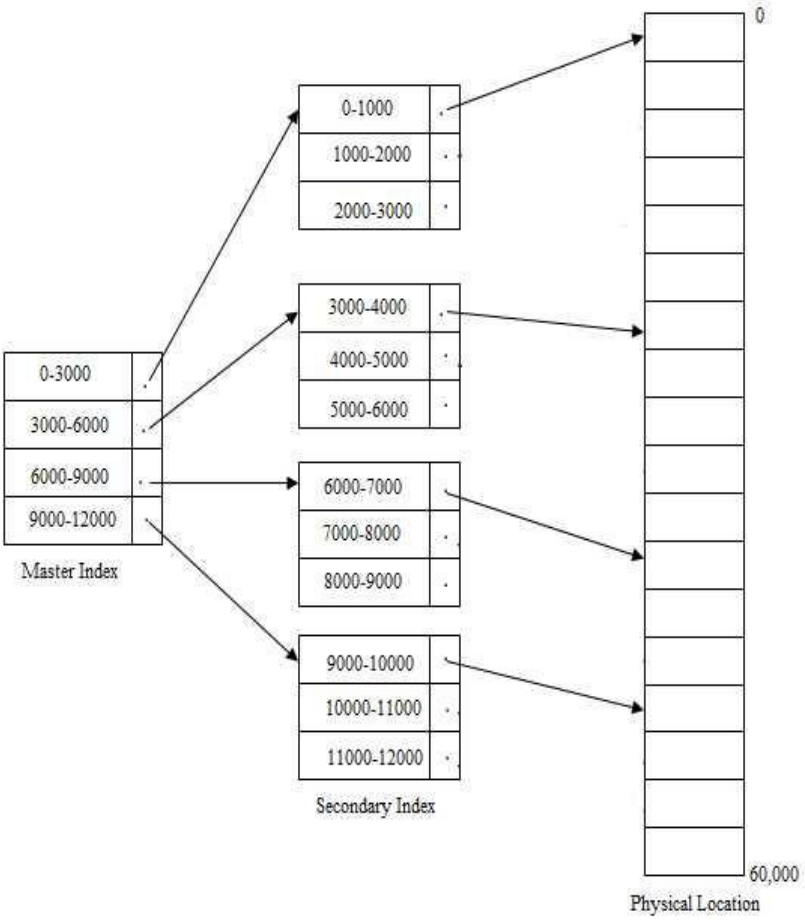


Fig 4.12 Indexed Sequential file

4.2.3 FILE DIRECTORIES

Directory structure

The file systems of computers can be extensive. Some systems store millions of files on terabytes of disk. To manage all these data, we need to organize them. This organization is usually done in two parts.

First, disks are split into one or more partitions. Typically, each disk on a system contains at least one partition, which is a low-level structure in which files and directories reside. Second, each partition contains information about files within it. This information is kept in entries in a device directory or volume table of contents. The device directory records information—such as name, location, size, and type—for all files on that partition.

The directory can be viewed as a symbol table that translates file names into their directory entries. We want to be able to insert entries, to delete entries, to search for a named entry, and to list all the entries in the directory.

Operations on file directories:

When considering a particular directory structure, following are the operations that are to be performed on a directory:

- Search for a file: We need to be able to search a directory structure to find the entry for a particular file. Since files have symbolic names and similar names may indicate a relationship between files, we may want to be able to find all files whose names match a particular pattern.
- Create a file: Make an entry in the directory when a file is created.
- Delete a file: When a file is no longer needed, we want to 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: Because the name of a file represents its contents to its users, we must be able to change the name when the contents or use of the file changes. Renaming a file may also allow its position within the directory structure to be changed.
- Traverse the file system: We may wish to access every directory and every file within a directory structure. For reliability, it is a good idea to save the contents and structure of the entire file system at regular intervals. Often, we do this by copying all files to magnetic tape. This technique provides a backup copy in case of system failure. In addition, if a file is no longer in use, the file can be copied to tape and the disk space of that file released for reuse by another file.

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, however, when the number of files increases or when the system has more than one user. Since all files are in the same directory, they must have unique names.

Even a single user on a single-level directory may find it difficult to remember the names of all the files as the number of files increases. It is not uncommon for a user to have hundreds of files on one computer system and an equal number of additional files on another system. Keeping track of so many files is a daunting task.

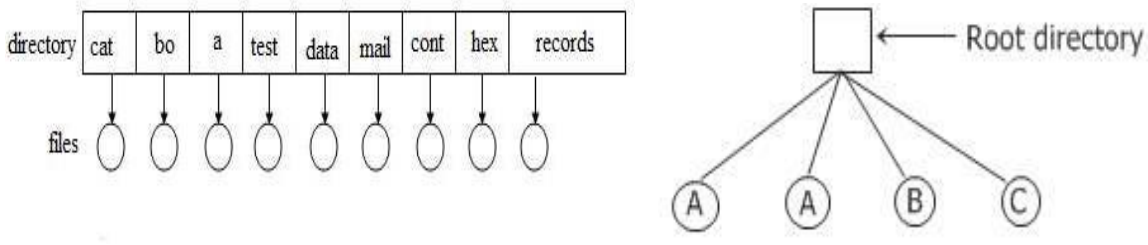


Fig 4.13 Single-level directory

Two-Level Directory

As we have seen, a single-level directory often leads to confusion of file names between different users. The standard solution is to create a separate directory for each user.

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

The user directories themselves must be created and deleted as necessary. A special system program is run with the appropriate user name and account information. The program creates a new UFD and adds an entry for it to the MFD. The execution of this program might be restricted to system administrators.

Although the two-level directory structure solves the name- collision problem, it still has disadvantages. This structure effectively isolates one user from another. This isolation is an advantage when the users are completely independent but is a disadvantage when the users want to cooperate on some task and to access one another's files. Some systems simply do not allow local user files to be accessed by other users

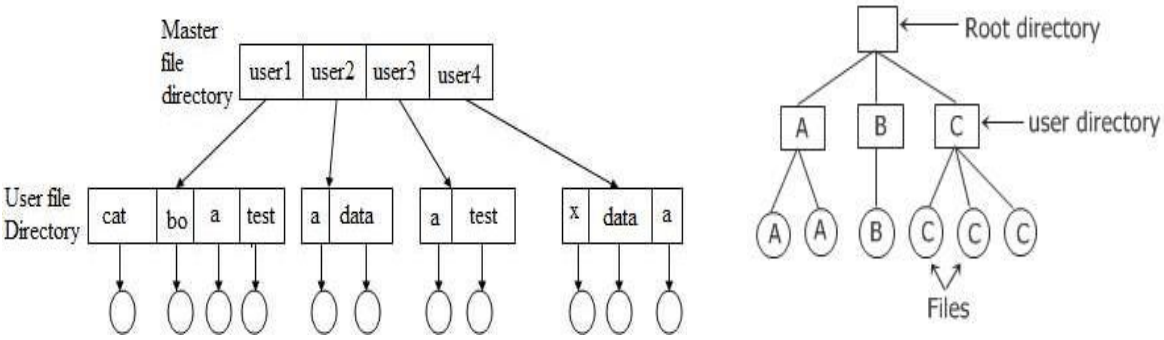


Fig.4.14 Two level directory structure.

Hierarchical Directory System

Hierarchical directory system is used for users with a large number of files, as the single-level directory system and two-level directory system is not satisfactory. The two-level directory system even on a single-user PC, is inconvenient. Since it is common for the users, want to group their files together in logical ways. Therefore, some way is needed is a general hierarchy. Here, hierarchy means a tree of directories. With hierarchical approach, each user on the computer system can have as many directories as are needed, so that files can be grouped together in natural ways.

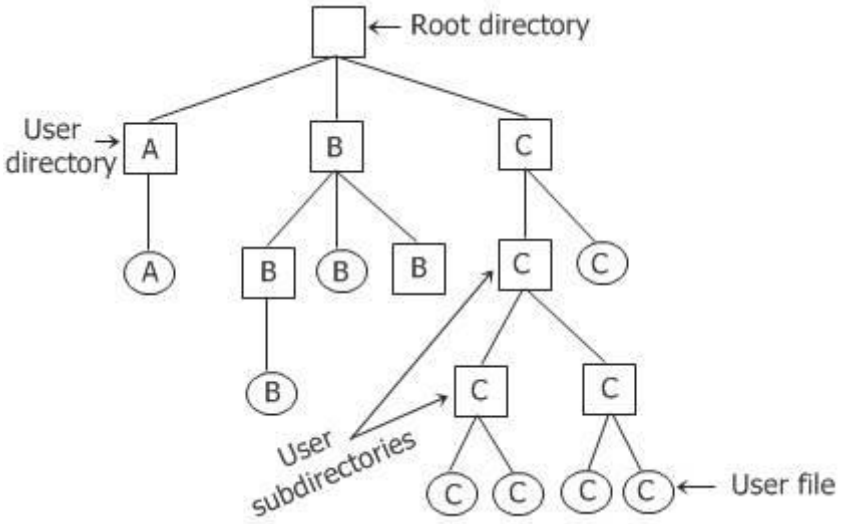


Fig.4.15 Hierarchical directory structure

General graph Directory

A serious problem with using an acyclic-graph structure is ensuring that there are no cycles. If we start with a two-level directory and allow users to create subdirectories, a tree-structured directory results. It should be fairly easy to see that simply adding new files and subdirectories to an existing tree-structured directory preserves the tree-structured nature. However, when we add links to an existing tree-structured directory, the tree structure is destroyed, resulting in a simple graph structure.

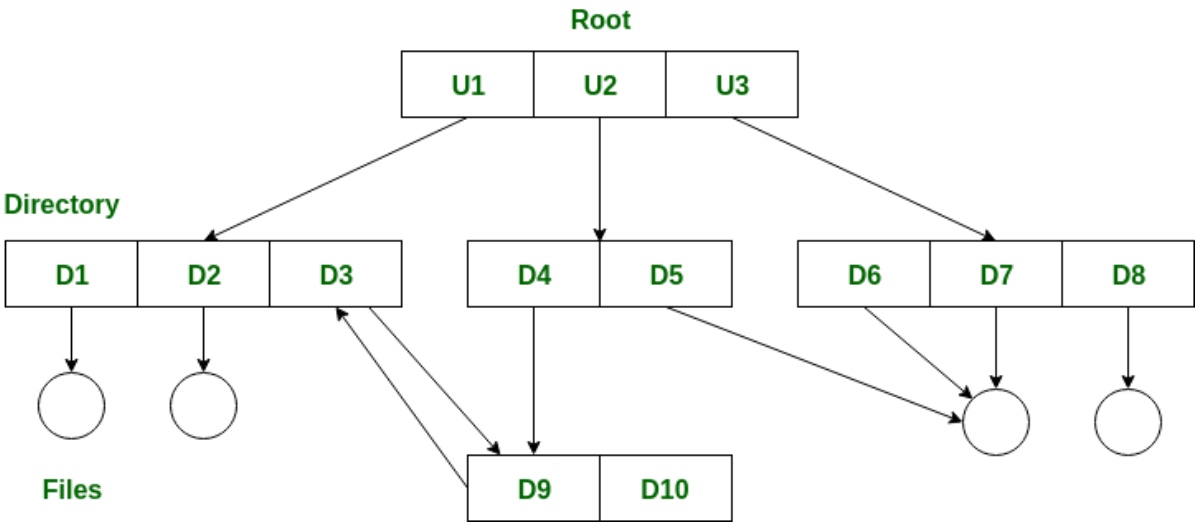


Fig.4.16 General graph directory structure

4.2.4 FILE ALLOCATION METHODS

The direct-access nature of disks allows us flexibility in the implementation of files. In almost every case, many files are stored on the same disk. The main problem is how to allocate space to these files so that disk space is utilized effectively and files can be accessed quickly. Three major methods of allocating disk space are in wide use: contiguous, linked, and indexed. Each method has advantages and disadvantages. Some systems support all three. More commonly, a system uses one method for all files within a file-system type.

Contiguous allocation method:

The contiguous-allocation method requires each file to occupy a set of contiguous blocks on the disk. Disk addresses define a linear ordering on the disk. With this ordering, assuming that only one job is accessing the disk, accessing block *b* + 1after block *b* normally requires no head movement. When head movement is needed (from the last sector of one cylinder to the first sector of the next cylinder), it is only one track. Thus, the number of disk seeks required for accessing contiguously allocated files is minimal.

Contiguous allocation of a file is defined by the disk address of the first block and length (in block units). If the file is n blocks long and starts at location b, then it occupies blocks b, b + 1, b + 2,…, b + n − 1. The directory entry for each file indicates the address of the starting block and the length of the area allocated for this file.

Accessing a file that has been allocated contiguously is easy. For sequential access, the file system remembers the disk address of the last block referenced and, when necessary, reads the next block. For direct access to block i of a file that starts at block b, we can immediately access block b + i. Thus, both sequential and direct access can be supported by contiguous allocation

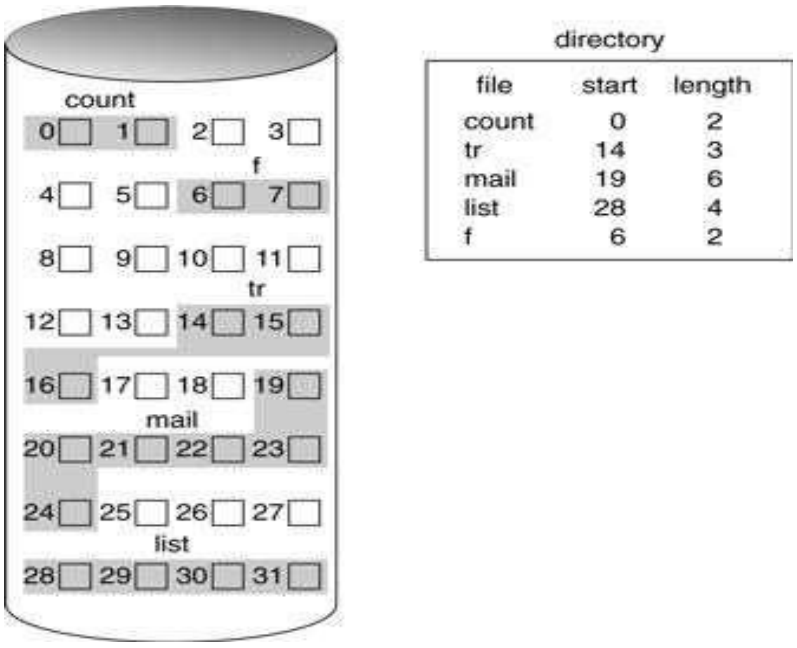


Fig 4.17 Contiguous allocation of disk space

One difficulty with contiguous allocation is finding space for anew file. To solve this problem dynamic allocation strategy is used. Here for contiguous free blocks we use a word _hole‘. There are three methods in this strategy.

- First fit: It allocates the first hole that is big enough. Searching can be starts from the beginning of the disk, or where the last first fit search ended. We can stop searching as soon as we find a large enough free hole.

- Best fit: It allocates the smallest hole, which is big enough. Searching can be starts from the beginning of the disk, it searches the entire disk and allocates the hole that is big enough for the required file.

- Worst fit: It allocates the largest hole that is big enough. Searching can be started from the beginning of the disk, it searches the entire disk and allocated the largest hole that is big enough for the given file.

Simulations have shown that both first fit and best fit are more efficient than worst fit in terms of both time and storage utilization. Neither first fit nor best fit is clearly best in terms of storage utilization, but first fit is generally faster.

These algorithms suffer from the problem of external fragmentation. As files are allocated and deleted, the free disk space is broken into little pieces. External fragmentation exists when enough total disk space exists to satisfy a request, but it is not contiguous; storage is fragmented into a number of holes, no one of which is large enough to store the data.

Another problem with contiguous allocation is determining how much space is needed for a file. When the file is created, the total amount of space it will need must be found and allocated. How does the creator (program or person) know the size of the file to be created?

Linked allocation

Linked allocation solves all problems of contiguous allocation. With linked allocation, each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and last blocks of the file. For example, a file of five blocks might start at block 9, continue at block 16, then block 1, block 10, and finally block 25 as shown in figure. Each block contains a pointer to the next block. Thesepointers are not made available to the user. Thus, if each block is 512 bytes, and a disk address (the pointer) requires 4 bytes, thenthe user sees blocks of 508 bytes.

To create a new file, we simply create a new entry in the directory. With linked allocation, each directory entry has a pointerto the first disk block of the file. This pointer is initialized to *nil* (the end-of-list pointer value) to signify an empty file. The size field is also set to 0. A write to the file causes a free block to be found via the free-space-management system, and this new blockis then written to, and is linked to the end of the file. To read a file,we simply read blocks by following the pointers from block to block.

There is no external fragmentation with linked allocation, and any free block on the free-space list can be used to satisfy arequest. The size of a file does not need to be declared when that file is created. A file can continue to grow as long as free blocksare available.

The major problem with linked allocation is that it can be usedeffectively only for sequential-access files. To find the *i*th block of a file, we must start at the beginning of that file, and follow the pointers until we get to the *i*th block.

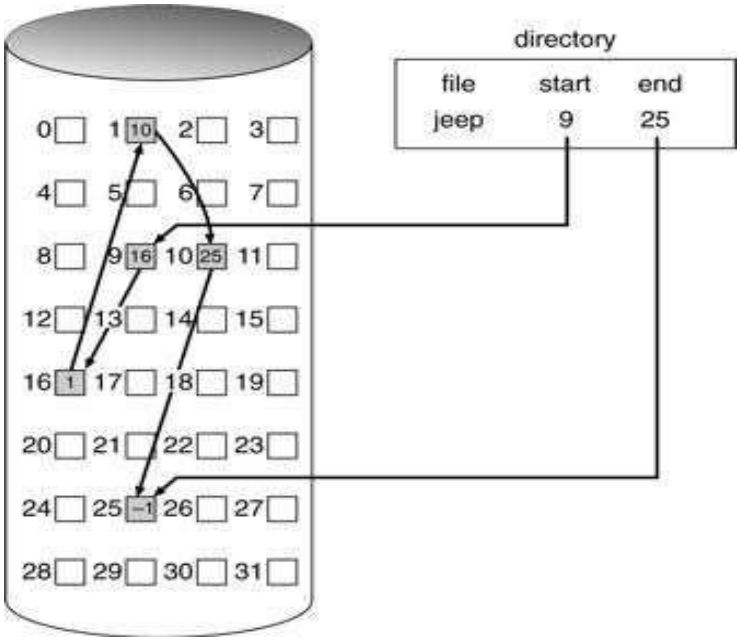


Fig 4.18 Linked allocation of disk space

Another disadvantage to linked allocation is the space required for the pointers. If a pointer requires 4 bytes out of a 512.byte block, then 0.78 percent of the disk is being used for pointers, rather than for information. Each file requires slightly more space.

Yet another problem of linked allocation is reliability. Since the files are linked together by pointers scattered all over the disk, if a pointer were lost or damaged, a bug in the operating-system software or a disk hardware failure might result in picking up the wrong pointer. This error could result in linking into the free-space list or into another file.

Grouped or Indexed allocation

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

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

When the file is created, all pointers in the index block are set to *nil*. When the *i*th block is first written, a block is obtained from the free-space manager, and its address is put in the *i*th index-block entry.

Indexed allocation supports direct access, without suffering from external fragmentation, because any free block on the disk may satisfy a request for more space Indexed allocation does suffer from wasted space. The pointer overhead of the index block is generally greater than the pointer overhead of linked allocation. Consider a

common case in which we have a file of only one or two blocks. With linked allocation,we lose the space of only one pointer per block (one or two pointers). With indexed allocation, an entire index block must be allocated, even if only one or two pointers will be non-*nil*.

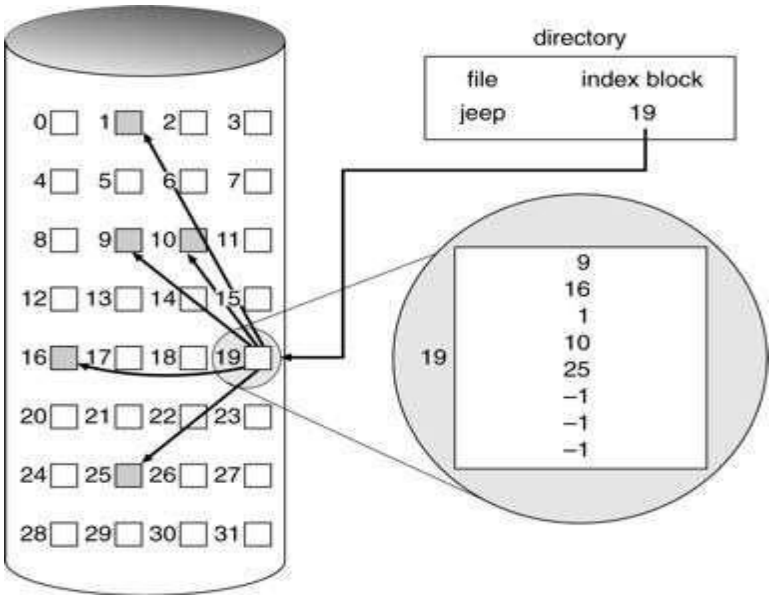


Fig 4.19 Indexed allocation of disk block

This point raises the question of how large the index block should be. Every file must have an index block, so we want the index block to be as small as possible. If the index block is too small, however, it will not be able to hold enough pointers. An index block is normally one disk block. Thus, it can be read and written directly by itself. To allow for large files, we may link together several index blocks.

To allow for large files, several index files may be linked together. the linked representation is to use a first level index block to point to a set of second-level index blocks, which in turn point to the file blocks. To access a block, the operating system uses the first-level index to find a second-level index block, and that block to find the desired data block. This approach could be continued to a third or fourth level, depending on the desired maximum file size.

4.2.5 FILE SPACE MANAGEMENT

The allocation methods define how the files are stored in the disk blocks. There are three main disk space or file allocation methods – Contiguous Allocation, Linked Allocation, Indexed Allocation. The main idea behind these methods is to provide – Efficient disk space utilization, Fast access to the file blocks.

1. Contiguous Allocation

In this scheme, each file occupies a contiguous set of blocks on the disk. For example, if a file requires n blocks and is given a block b as the starting location, then the blocks assigned to the file will be: *b, b+1, b+2,……b+n-1*. This means that given the starting block address and the length of the file (in terms of blocks required), we can determine the blocks occupied by the file.

The directory entry for a file with contiguous allocation contains

- Address of starting block
- Length of the allocated portion.

The *file 'mail'* in the following figure starts from the block 19 with length = 6 blocks. Therefore, it occupies *19, 20, 21, 22, 23, 24* blocks.

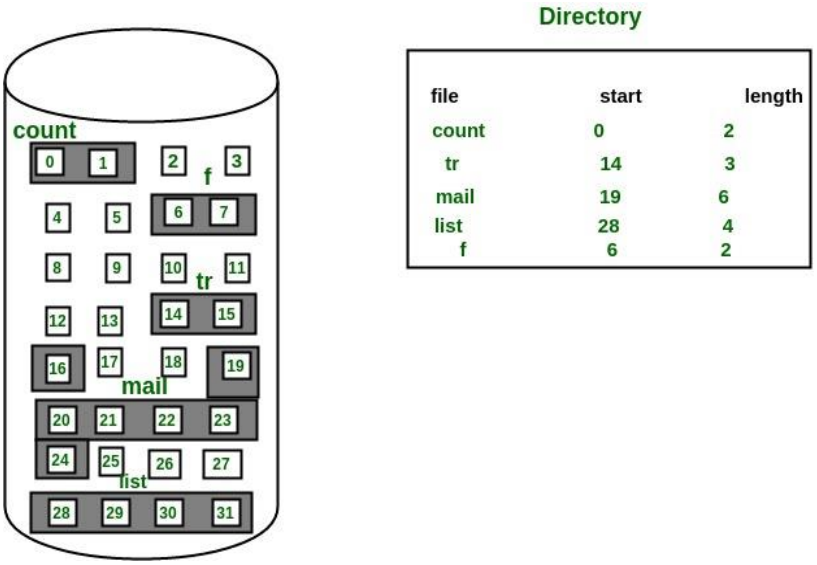


Fig 4.20 Contiguous File Allocation

Advantages:

- Both the Sequential and Direct Accesses are supported by this. For direct access, the address of the kth block of the file which starts at block b can easily be obtained as (b+k).
- This is extremely fast since the number of seeks are minimal because of contiguous allocation of file blocks.

Disadvantages:

- This method suffers from both internal and external fragmentation. This makes it inefficient in terms of memory utilization.
- Increasing file size is difficult because it depends on the availability of contiguous memory at a particular instance.

2. Linked List Allocation

In this scheme, each file is a linked list of disk blocks which need not be contiguous. The disk blocks can be scattered anywhere on the disk. The directory entry contains a pointer to the starting and the ending file block. Each block contains a pointer to the next block occupied by the file. The file 'jeep' in following image shows how the blocks are randomly distributed. The last block (25) contains -1 indicating a null pointer and does not point to any other block.

Advantages:

- This is very flexible in terms of file size. File size can be increased easily since the system does not have to look for a contiguous chunk of memory.

- This method does not suffer from external fragmentation. This makes it relatively better in terms of memory utilization.

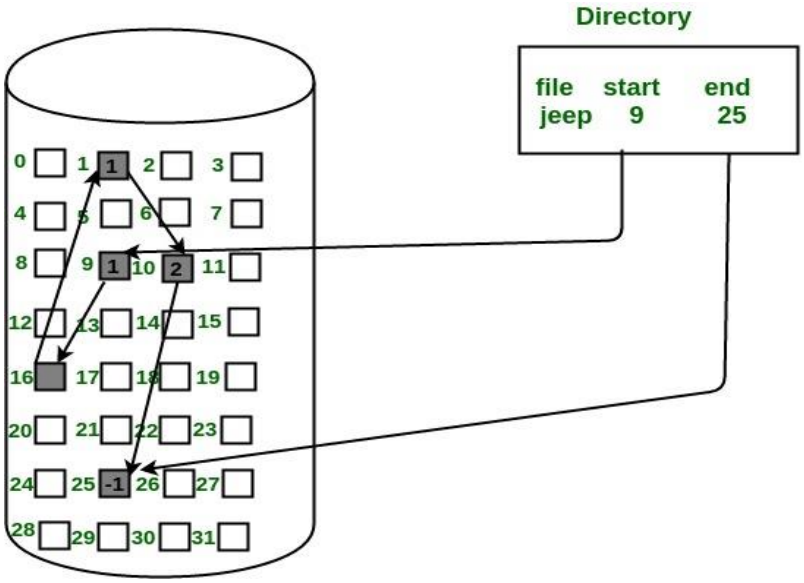


Fig 4.21 Linked List Allocation

Disadvantages:

- Because the file blocks are distributed randomly on the disk, a large number of seeks are needed to access every block individually. This makes linked allocation slower.
- It does not support random or direct access. We cannot directly access the blocks of a file. A block k of a file can be accessed by traversing k blocks sequentially (sequential access) from the starting block of the file via block pointers.
- Pointers required in the linked allocation incur some extra overhead.

3. Indexed Allocation

In this scheme, a special block known as the **Index block** contains the pointers to all the blocks occupied by a file. Each file has its own index block. The ith entry in the index block contains the disk address of the ith file block. The directory entry contains the address of the index block as shown in the image:

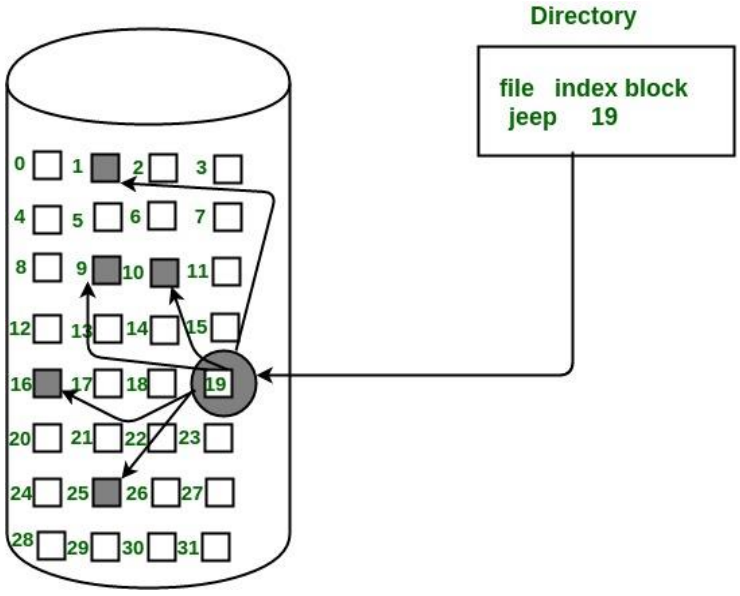


Fig 4.22 Indexed Allocation Method

Advantages:

- This supports direct access to the blocks occupied by the file and therefore provides fast access to the file blocks.
- It overcomes the problem of external fragmentation.

Disadvantages:

- The pointer overhead for indexed allocation is greater than linked allocation.
- For very small files, say files that expand only 2-3 blocks, the indexed allocation would keep one entire block (index block) for the pointers which is inefficient in terms of memory utilization. However, in linked allocation we lose the space of only 1 pointer per block.

For files that are very large, single index block may not be able to hold all the pointers. Following mechanisms can be used to resolve this:

1. **Linked scheme:** This scheme links two or more index blocks together for holding the pointers. Every index block would then contain a pointer or the address to the next index block.

2. **Multilevel index:** In this policy, a first level index block is used to point to the second level index blocks which inturn points to the disk blocks occupied by the file. This can be extended to 3 or more levels depending on the maximum file size.

3. **Combined Scheme:** In this scheme, a special block called the Inode (information Node) contains all the information about the file such as the name, size, authority, etc and the remaining space of Inode is used to store the Disk Block addresses which contain the actual file *as shown in the image below.* The first few of these pointers in Inode point to the direct blocks i.e the pointers contain the addresses of the disk blocks that contain data of the file. The next few pointers point to indirect blocks. Indirect blocks may be single indirect, double indirect or triple indirect. Single Indirect block is the disk block that does not contain the file data but the disk address of the blocks that contain the file data. Similarly, double indirect blocks do not contain the file data but the disk address of the blocks that contain the address of the blocks containing the file data.

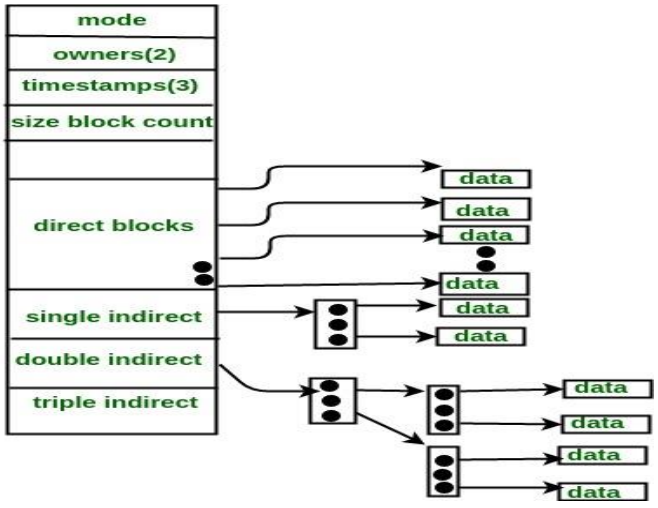


Fig 4.23 Inode Block

4.2.6 DISK SPACE MANAGEMENT

Since disk space is limited, we need to reuse the space from deleted files for new files, if possible. To keep track of free disk space, the system maintains a free-space list. The free-space list records all free disk blocks—those not allocated to some file or directory. To create a file, we search the free-space list for the required amount of space, and allocate that space to the new file. This space is then removed from the free-space list. When a file is deleted, its disk space is added to the free-space list. The free- space list, despite its name, might not be implemented as a list, as we shall discuss.

Bit Vector

Frequently, the free-space list is implemented as a bit map or bit vector. Each block is represented by 1 bit. If the block is free, the bit is 1; if the block is allocated, the bit is 0. For example, consider a disk where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 are free, and the rest of the blocks areallocated. The free-space bit map would be

0011110011111100011000000011100000 …

The main advantage of this approach is its relative simplicity and efficiency in finding the first free block, or consecutive free blocks on the disk.

Linked List

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

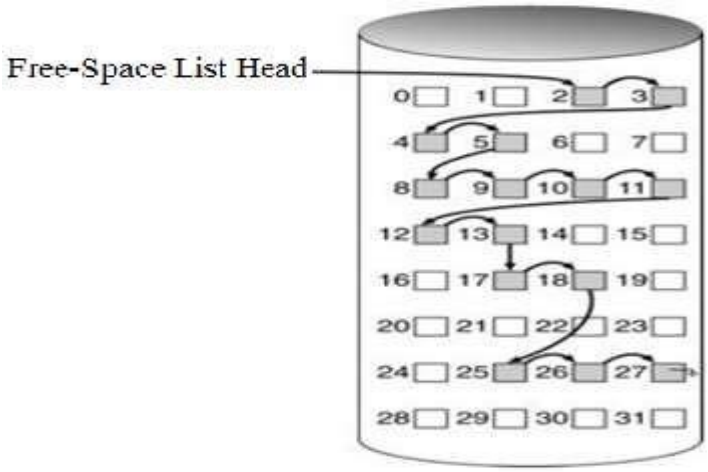


Fig 4.24 Linked free space list on disk

4.2.7 RECORD BLOCKING

Records are the logical unit of a file whereas blocks are units of I/O with secondary storage. It performs for I/O and the records must organize as blocks. Blocking is a process of grouping similar records into blocks that the operating system component explores

exhaustively. Block is the unit of I/O for secondary storage. Generally, the larger blocks reduce the I/O transfer time. Larger blocks require larger I/O buffers. In record blocking, the records are grouped into blocks by shared properties that are indicators of duplication.

Generally, there are three types of record blocking methods – Fixed blocking, Variable-length spanned blocking, and Variable-length un-spanned blocking.

Fixed blocking

In this method, record lengths are fixed. The prescribed number of records stored in a block. Internal fragmentation is stored in a block. Fixed blocking is common for sequential files with fixed-length records length records

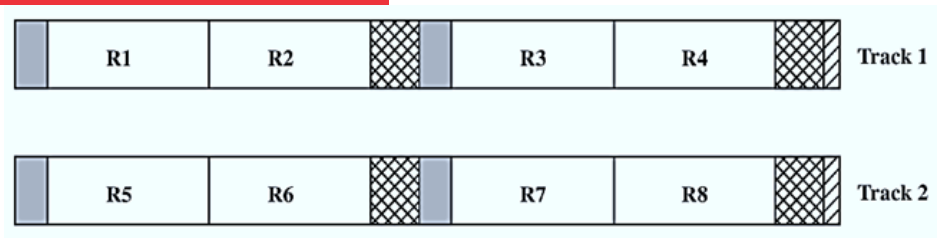


Fig 4.25 Fixed Record Blocking

Variable-length spanned blocking

In this method, record sizes aren't same, Variable-length records packing into blocks with no unused space. So, some records may divide into two blocks. In this type of situation, a pointer passes from one block to another block. So, the Pointers used to span blocks unused space. It is efficient in length and efficiency of storage. It doesn't limit record size, but it is more complicated to implement. So, the files are more difficult to update.

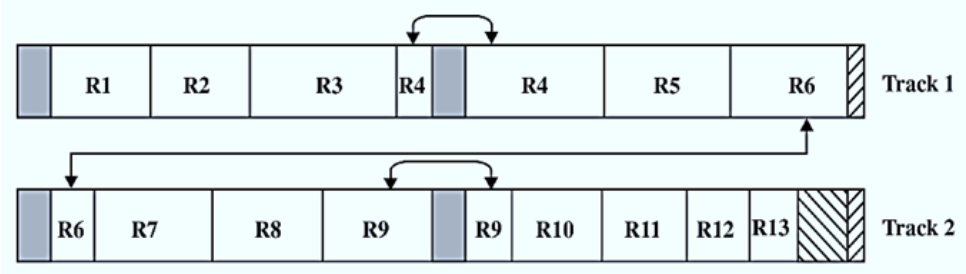


Fig 4.26 Variable-length spanned blocking

Variable-length un-spanned blocking

Here, records are variable in length, but the records span between blocks. In this method, the wasted area is a serious problem, because of the inability to use the remainder of a block, if the next record is larger than the remaining unused space. These blocking methods result in blocking results in wasted space and limits record size to the size of the block.

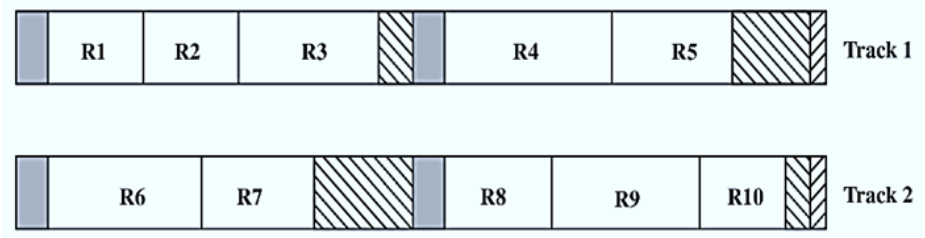


Fig 4.27 Variable-length unspanned blocking

4.3 PROTECTION MECHANISMS

4.3.1CRYPTOGRAPHY

Cryptography is technique of securing information and communications through use of codes so that only those persons for whom the information is intended can understand it and process it, thus preventing unauthorized access to information. The prefix “crypt” means “hidden” and suffix – “graphy” means “writing”.

In Cryptography the techniques which are used to protect information are obtained from mathematical concepts and a set of rule based calculations known as algorithms to convert messages in ways that "make it hard to decode it. These algorithms are used for cryptographic key generation, digital signing, verification to protect data privacy, web browsing on internet and to protect confidential transactions such as credit card and debit card transactions.

Techniques used For Cryptography

In today’s age of computers cryptography is often associated with the process where an ordinary plain text is converted to cipher text which is the text made such that intended receiver of the text can only decode it and hence this process is known as encryption. The process of conversion of cipher text to plain text this is known as decryption.

Features Of Cryptography

- Confidentiality: Information can only be accessed by the person for whom it is intended and no other person except him can access it.
- Integrity: Information cannot be modified in storage or transition between sender and intended receiver without any addition to information being detected.
- Non-repudiation: The creator/sender of information cannot deny his intention to send information at later stage.
- Authentication: The identities of sender and receiver are confirmed. As well as destination/origin of information is confirmed.

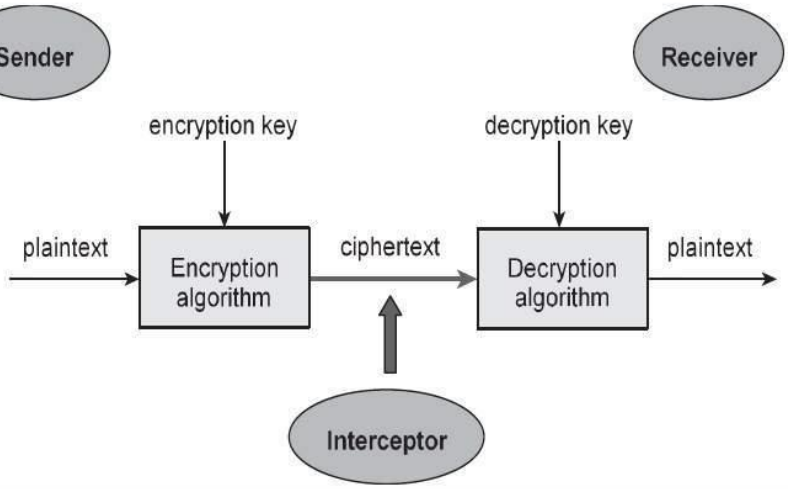


Fig 4.28 Encryption and Decryption of data

Components of a Cryptosystem

The various components of a basic cryptosystem are as follows −

- Plaintext. It is the data to be protected during transmission.
- Encryption Algorithm. It is a mathematical process that produces a ciphertext for any given plaintext and encryption key. It is a cryptographic algorithm that takes plaintext and an encryption key as input and produces a ciphertext.
- Ciphertext. It is the scrambled version of the plaintext produced by the encryption algorithm using a specific the encryption key. The ciphertext is not guarded. It flows on public channel. It can be intercepted or compromised by anyone who has access to the communication channel.
- Decryption Algorithm, It is a mathematical process, that produces a unique plaintext for any given ciphertext and decryption key. It is a cryptographic algorithm that takes a ciphertext and a decryption key as input, and outputs a plaintext. The decryption algorithm essentially reverses the encryption algorithm and is thus closely related to it.
- Encryption Key. It is a value that is known to the sender. The sender inputs the encryption key into the encryption algorithm along with the plaintext in order to compute the ciphertext.
- Decryption Key. It is a value that is known to the receiver. The decryption key is related to the encryption key, but is not always identical to it. The receiver inputs the decryption key into the decryption algorithm along with the ciphertext in order to compute the plaintext.

For a given cryptosystem, a collection of all possible decryption keys is called a key space.

An interceptor (an attacker) is an unauthorized entity who attempts to determine the plaintext. He can see the ciphertext and may know the decryption algorithm. He, however, must never know the decryption key.

Types of Cryptosystems

Fundamentally, there are two types of cryptosystems based on the manner in which encryption-decryption is carried out in the system − Symmetric Key Encryption, Asymmetric Key Encryption The main difference between these cryptosystems is the relationship between the encryption and the decryption key. Logically, in any cryptosystem, both the keys are closely associated. It is practically impossible to decrypt the ciphertext with the key that is unrelated to the encryption key.

Symmetric Key Encryption

The encryption process where same keys are used for encrypting and decrypting the information is known as Symmetric Key Encryption. The study of symmetric cryptosystems is referred to as symmetric cryptography. Symmetric cryptosystems are also sometimes referred to as secret key cryptosystems.

A few well-known examples of symmetric key encryption methods are − Digital Encryption Standard (DES), Triple-DES (3DES), IDEA, and BLOWFISH.

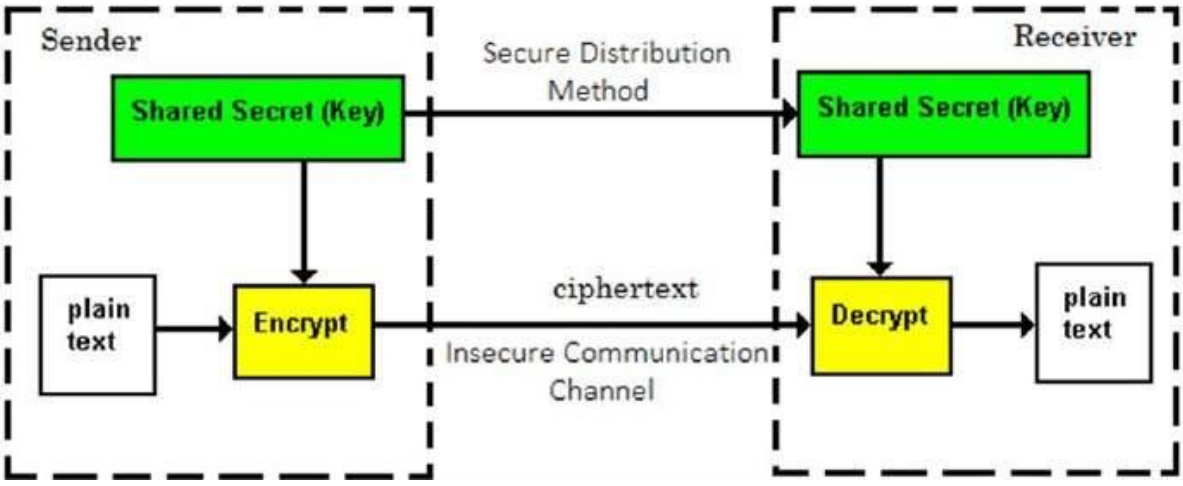


Fig 4.29 Symmetric key encryption

Prior to 1970, all cryptosystems employed symmetric key encryption. Even today, its relevance is very high and it is being used extensively in many cryptosystems. It is very unlikely that this encryption will fade away, as it has certain advantages over asymmetric key encryption.

Salient features

- Persons using symmetric key encryption must share a common key prior to exchange of information.
- Keys are recommended to be changed regularly to prevent any attack on the system.
- A robust mechanism needs to exist to exchange the key between the communicating parties. As keys are required to be changed regularly, this mechanism becomes expensive and cumbersome.
- In a group of n people, to enable two-party communication between any two persons, the number of keys required for group is $n \times (n - 1)/2$.
- Length of Key (number of bits) in this encryption is smaller and hence, process of encryption-decryption is faster than asymmetric key encryption.
- Processing power of computer system required to run symmetric algorithm is less.

Challenge of Symmetric Key Cryptosystem

There are two restrictive challenges of employing symmetric key cryptography.

- Key establishment − Before any communication, both the sender and the receiver need to agree on a secret symmetric key. It requires a secure key establishment mechanism in place.
- Trust Issue − Since the sender and the receiver use the same symmetric key, there is an implicit requirement that the sender and the receiver 'trust' each other. For example, it may happen that the receiver has lost the key to an attacker and the sender is not informed.

These two challenges are highly restraining for modern day communication. Today, people need to exchange information with non-familiar and non-trusted parties, for example, a communication between online seller and customer. These limitations of symmetric key encryption gave rise to asymmetric key encryption schemes.

Asymmetric Key Encryption

The encryption process where different keys are used for encrypting and decrypting the information is known as Asymmetric Key Encryption. Though the keys are different, they are mathematically related and hence, retrieving the plaintext by decrypting ciphertext is feasible. The process is depicted in the following illustration −

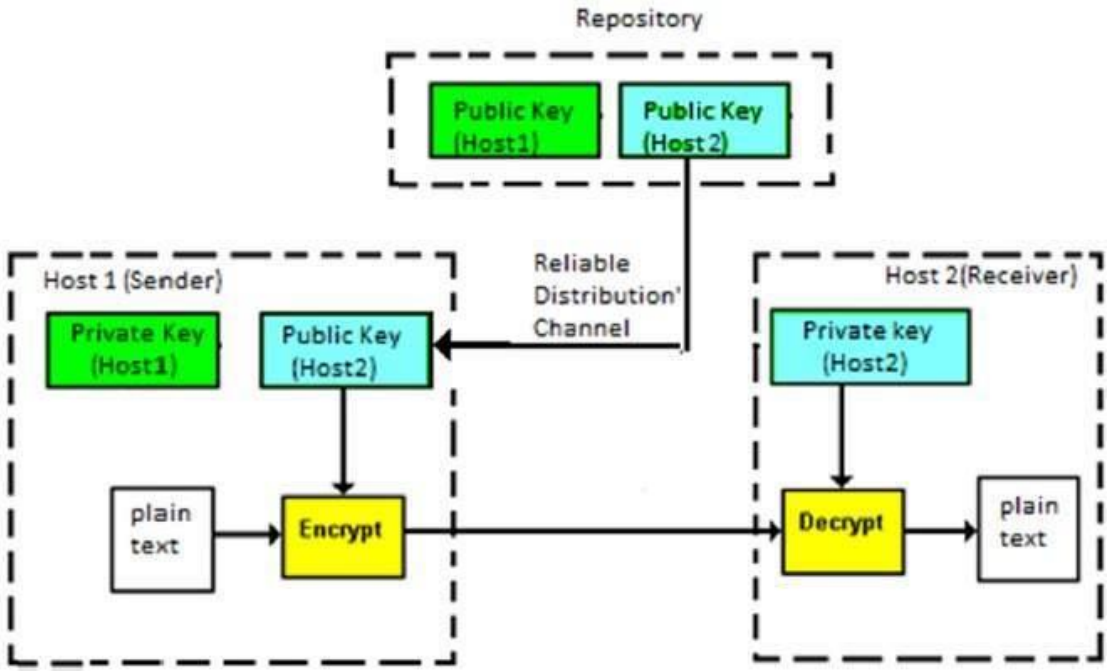


Fig 4.30 Asymmetric key encryption

Asymmetric Key Encryption was invented in the 20th century to come over the necessity of pre-shared secret key between communicating persons

Salient features

- Every user in this system needs to have a pair of dissimilar keys, private key and public key. These keys are mathematically related − when one key is used for encryption, the other can decrypt the ciphertext back to the original plaintext.
- It requires to put the public key in public repository and the private key as a well-guarded secret. Hence, this scheme of encryption is also called Public Key Encryption.
- Though public and private keys of the user are related, it is computationally not feasible to find one from another. This is a strength of this scheme.
- When *Host1* needs to send data to *Host2*, he obtains the public key of *Host2* from repository, encrypts the data, and transmits.
- *Host2* uses his private key to extract the plaintext.
- Length of Keys (number of bits) in this encryption is large and hence, the process of encryption-decryption is slower than symmetric key encryption.
- Processing power of computer system required to run asymmetric algorithm is higher.

Symmetric cryptosystems are a natural concept. In contrast, public-key cryptosystems are quite difficult to comprehend.

You may think, how can the encryption key and the decryption key are 'related', and yet it is impossible to determine the decryption key from the encryption key? The answer lies in the mathematical concepts. It is possible to design a cryptosystem whose keys have this property. The concept of public-key cryptography is relatively new. There are fewer public-key algorithms known than symmetric algorithms.

Challenge of Public Key Cryptosystem

Public-key cryptosystems have one significant challenge − the user needs to trust that the public key that he is using in communications with a person really is the public key of that person and has not been spoofed by a malicious third party.

This is usually accomplished through a Public Key Infrastructure (PKI) consisting a trusted third party. The third party securely manages and attests to the authenticity of public keys. When the third party is requested to provide the public key for any communicating person X, they are trusted to provide the correct public key.

The third party satisfies itself about user identity by the process of attestation, notarization, or some other process − that X is the one and only, or globally unique, X. The most common method of making the verified public keys available is to embed them in a certificate which is digitally signed by the trusted third party.

Relation between Encryption Schemes

A summary of basic key properties of two types of cryptosystems is given below −

	Symmetric Cryptosystems	Public Key Cryptosystems
Relation between Keys	Same	Different, but mathematically related
Encryption Key	Symmetric	Public
Decryption Key	Symmetric	Private

Due to the advantages and disadvantage of both the systems, symmetric key and public-key cryptosystems are often used together in the practical information security systems.

4.3.2DIGITAL SIGNATURE

A digital signature is a mathematical technique used to validate the authenticity and integrity of a message, software, or digital document.

1. **Key Generation Algorithms**: Digital signature is electronic signatures, which assure that the message was sent by a particular sender. While performing digital transactions authenticity and integrity should be assured, otherwise, the data can be altered or someone can also act as if he was the sender and expect a reply.
2. **Signing Algorithms**: To create a digital signature, signing algorithms like email programs create a one-way hash of the electronic data which is to be signed. The signing algorithm then encrypts the hash value using the private key (signature key).

This encrypted hash along with other information like the hashing algorithm is the digital signature. This digital signature is appended with the data and sent to the verifier. The reason for encrypting the hash instead of the entire message or document is that a hash function converts any arbitrary input into a much shorter fixed-length value. This saves time as now instead of signing a long message a shorter hash value has to be signed and moreover hashing is much faster than signing.

3. **Signature Verification Algorithms** : Verifier receives Digital Signature along with the data. It then uses Verification algorithm to process on the digital signature and the public key (verification key) and generates some value. It also applies the same hash function on the received data and generates a hash value. Then the hash value and the output of the verification algorithm are compared. If they both are equal, then the digital signature is valid else it is invalid.

The steps followed in creating digital signature are :

1. Message digest is computed by applying hash function on the message and then message digest is encrypted using private key of sender to form the digital signature. (digital signature = encryption (private key of sender, message digest) and message digest = message digest algorithm(message)).
2. Digital signature is then transmitted with the message.(message + digital signature is transmitted)
3. Receiver decrypts the digital signature using the public key of sender.(This assures authenticity, as only sender has his private key so only sender can encrypt using his private key which can thus be decrypted by sender's public key).
4. The receiver now has the message digest.
5. The receiver can compute the message digest from the message (actual message is sent with the digital signature).
6. The message digest computed by receiver and the message digest (got by decryption on digital signature) need to be same for ensuring integrity.

Message digest is computed using one-way hash function, i.e. a hash function in which computation of hash value of a message is easy but computation of the message from hash value of the message is very difficult.

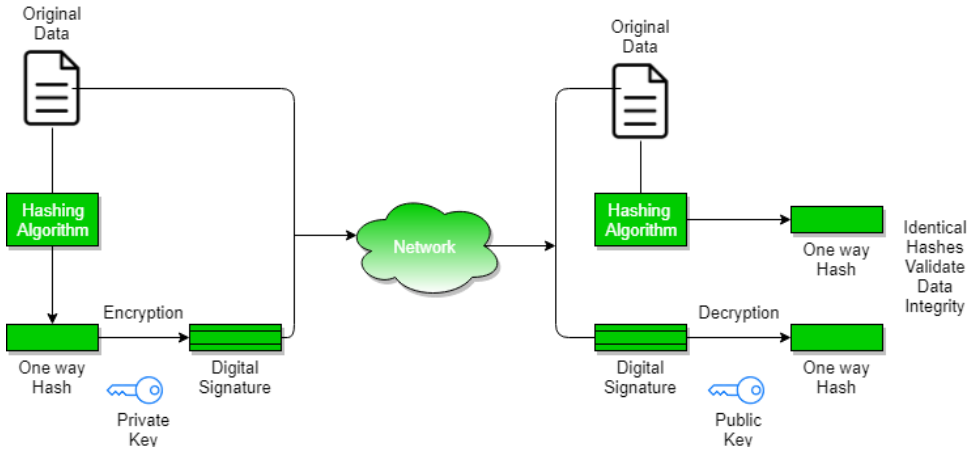


Fig 4.31Digital Signature

4.3.3 USER AUTHENTICATION

The main objective of authentication is to allow authorized users to access the computer and to deny access to unauthorized users. Operating Systems generally identify/authenticates users using the following 3 ways: Passwords, Physical identification, and Biometrics. These are explained as following below.

Passwords: Password verification is the most popular and commonly used authentication technique. A password is a secret text that is supposed to be known only to a user. In a password-based system, each user is assigned a valid username and password by the system administrator. The system stores all usernames and Passwords. When a user logs in, their user name and password are verified by comparing them with the stored login name and password. If the contents are the same then the user is allowed to access the system otherwise it is rejected.

Physical Identification: This technique includes machine-readable badges(symbols), cards, or smart cards. In some companies, badges are required for employees to gain access to the organization's gate. In many systems, identification is combined with the use of a password i.e the user must insert the card and then supply his /her password. This kind of authentication is commonly used with ATMs. Smart cards can enhance this scheme by keeping the user password within the card itself. This allows authentication without the storage of passwords in the computer system. The loss of such a card can be dangerous.

Biometrics: This method of authentication is based on the unique biological characteristics of each user such as fingerprints, voice or face recognition, signatures, and eyes.

QUESTIONS

1. Explain DMA
2. Explain in detail Disk I/O
3. What are the design considerations for Disk I/O
4. Explain Cryptography
5. Explain Free Space Management
6. Elaborate on File Space Management