UNIT - I

NEED AND FUNCTIONS OF OPERATING SYSTEMS

Goal of an operating system:

The fundamental goal of a Computer System is to execute user programs and to make tasks easier. Various application programs along with hardware system are used to perform this work. Operating System is a software which manages and controls the entire set of resources and effectively utilize every part of a computer.

The figure shows how OS acts as a medium between hardware unit and application programs.

Need of Operating System:

OS as a platform for Application programs:

Operating system provides a platform, on top of which, other programs, called application programs can run. These application programs help the users to perform a specific task easily. It acts as an interface between the computer and the user. It is designed in such a manner that it operates, controls and executes various applications on the computer.

Managing Input-Output unit:

Operating System also allows the computer to manage its own resources such as memory, monitor, keyboard, printer etc. Management of these resources is required for an effective utilization. The operating system controls the various system input-output resources and allocates them to the users or programs as per their requirement.

Consistent user interface:

Operating System provides the user an easy-to-work user interface, so the user doesn't have to learn a different UI every time and can focus on the content and be productive as quickly as possible. Operating System provides templates, UI components to make the working of a computer, really easy for the user.

Multitasking:

Operating System manages memory and allow multiple programs to run in their own space and even communicate with each other through shared memory. Multitasking gives users a good experience as they can perform several tasks on a **Functions of an Operating System**

An operating system has variety of functions to perform. Some of the prominent functions of an operating system can be broadly outlined as:

Processor Management:

This deals with management of the Central Processing Unit (CPU). The operating system takes care of the allotment of CPU time to different processes. When a process finishes its CPU processing after executing for the allotted time period, this is called scheduling. There are various types of scheduling techniques that are used by the operating systems.

Device Management:

The Operating System communicates with hardware and the attached devices and maintains a balance between them and the CPU. This is all the more important because the CPU processing speed is much higher than that of I/O devices. In order to optimize the CPU time, the operating system employs two techniques Buffering and Spooling.

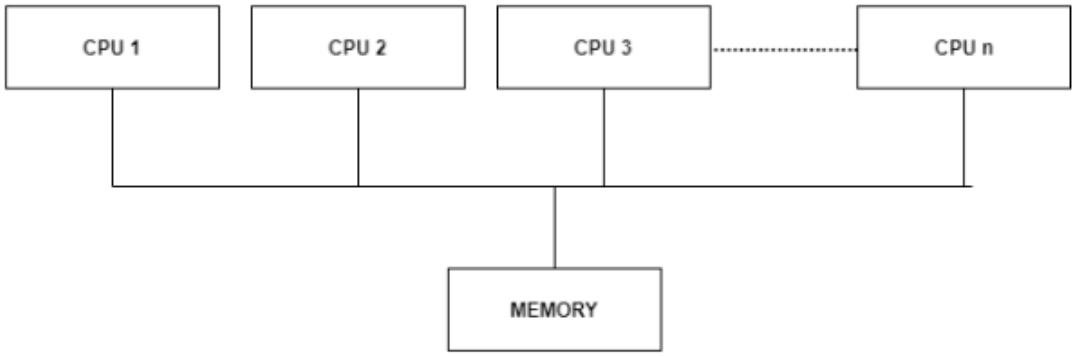
Buffering:

In this technique, input and output data is temporarily stored in Input Buffer and Output Buffer. Once the signal for input or output is sent to or from the CPU respectively, the operating system through the device controller moves the data from the input device to the input buffer and for the output device to the output buffer. In case of input, if the buffer is full, the operating system sends a signal to the program which processes the data stored in the buffer.

When the buffer becomes empty, the program informs the operating system which reloads the buffer and the input operation continues

Multiprocessor Systems

Most computer systems are single processor systems i.e they only have one processor. However, multiprocessor or parallel systems are increasing in importance nowadays. These systems have multiple processors working in parallel that share the computer clock, memory, bus, peripheral devices etc. An image demonstrating the multiprocessor architecture is:



Multiprocessing Architecture

Advantages of Multiprocessor Systems

There are multiple advantages to multiprocessor systems. Some of these are:

More reliable Systems

In a multiprocessor system, even if one processor fails, the system will not halt. This ability to continue working despite hardware failure is known as graceful degradation. For example: If there are 5 processors in a multiprocessor system and one of them fails, then also 4 processors are still working. So the system only becomes slower and does not ground to a halt.

Enhanced Throughput

If multiple processors are working in tandem, then the output of the system increases i.e. number of processes getting executed per unit of time increases. If there are N processors then the throughput increases by an amount just under N.

More Economic Systems

Multiprocessor systems are cheaper than single processor systems in the long run because they share the data storage, peripheral devices, power supplies etc. If there are

multiple processes that share data, it is better to schedule them on multiprocessor systems with shared data than have different computer systems with multiple copies of the data.

Distributed operating System

Distributed systems use multiple central processors to serve multiple real-time applications and multiple users. Data processing jobs are distributed among the processors accordingly.

The processors communicate with one another through various communication lines (such as high-speed buses or telephone lines). These are referred as **loosely coupled systems** or distributed systems. Processors in a distributed system may vary in size and function. These processors are referred as sites, nodes, computers, and so on.

The advantages of distributed systems are as follows –

- With resource sharing facility, a user at one site may be able to use the resources available at another.
- Speedup the exchange of data with one another via electronic mail.
- If one site fails in a distributed system, the remaining sites can potentially continue operating.
- Better service to the customers.
- Reduction of the load on the host computer.
- Reduction of delays in data processing.

Clustered Systems

Clustered systems are similar to parallel systems as they both have multiple CPUs. However, a major difference is that clustered systems are created by two or more individual computer systems merged together. Basically, they have independent computer systems with a common storage and the systems work together.

The clustered systems are a combination of hardware clusters and software clusters. The hardware clusters help in sharing of high performance disks between the systems. The software clusters makes all the systems work together.

Each node in the clustered systems contains the cluster software. This software monitors the cluster system and makes sure it is working as required. If any one of the nodes in the clustered system fail, then the rest of the nodes take control of its storage and resources and try to restart.

Types of Clustered Systems

There are primarily two types of clustered systems i.e. asymmetric clustering system and symmetric clustering system. Details about these are given as follows:

Asymmetric Clustering System

In this system, one of the nodes in the clustered system is in hot standby mode and all the others run the required applications. The hot standby mode is a failsafe in which a hot standby node is part of the system. The hot standby node continuously monitors the server and if it fails, the hot standby node takes its place.

Symmetric Clustering System

In symmetric clustering system two or more nodes all run applications as well as monitor each other. This is more efficient than asymmetric system as it uses all the hardware and doesnot keep a node merely as a hot standby.

Benefits of Clustered Systems

The difference benefits of clustered systems are as follows:

Performance:

Clustered systems result in high performance as they contain two or more individual computer systems merged together. These work as a parallel unit and result in much better performance for the system.

Fault Tolerance:

Clustered systems are quite fault tolerant and the loss of one node does not result in the loss of the system. They may even contain one or more nodes in hot standby mode which allows them to take the place of failed nodes.

Scalability:

Clustered systems are quite scalable as it is easy to add a new node to the system. There is no need to take the entire cluster down to add a new node.

Real Time operating System

A real-time system is defined as a data processing system in which the time interval required to process and respond to inputs is so small that it controls the environment. The time taken by the system to respond to an input and display of required updated information is termed as the **response time**. So, in this method, the response time is very less as compared to online processing.

Real-time systems are used when there are rigid time requirements on the operation of a processor or the flow of data and real-time systems can be used as a control device in a dedicated application. A real-time operating system must have well-defined, fixed time constraints, otherwise the system will fail. For example, Scientific experiments, medical imaging systems, industrial control systems, weapon systems, robots, air traffic control systems, etc.

There are two types of real-time operating systems.

Hard real-time systems:

Hard real-time systems guarantee that critical tasks complete on time. In hard real-time systems, secondary storage is limited or missing and the data is stored in ROM. In these systems, virtual memory is almost never found.

Soft real-time systems:

Soft real-time systems are less restrictive. A critical real-time task gets priority over other tasks and retains the priority until it completes. Soft real-time systems have limited utility than hard real-time systems. For example, multimedia, virtual reality, Advanced Scientific Projects like undersea exploration and planetary rovers, etc.

UNIT - II
PROCESS CONCEPTS

Process concepts

A process is basically a program in execution. The execution of a process must progress in a sequential fashion.

A process is defined as an entity which represents the basic unit of work to be implemented in the system.

Process Life Cycle

When a process executes, it passes through different states. These stages may differ in different operating systems, and the names of these states are also not standardized.
In general, a process can have one of the following five states at a time.

State & Description

This is the initial state when a process is first started / created.

Ready

The process is waiting to be assigned to a processor. Ready processes are waiting to have the processor allocated to them by the operating system so that they can run. Process may come into this state after **Start** state or while running it by but interrupted by the scheduler to assign CPU to some other process.

Running

Once the process has been assigned to a processor by the OS scheduler, the process state is set to running and the processor executes its instructions.

Waiting

Process moves into the waiting state if it needs to wait for a resource, such as waiting for user input, or waiting for a file to become available.

Terminated or Exit

The process is waiting to be assigned to a processor. Ready processes are waiting to have the processor allocated to them by the operating system so that they can run. Process may come into this state after **Start** state or while running it by but interrupted by the scheduler to assign CPU to some other process.

Process Control Block (PCB)

A Process Control Block is a data structure maintained by the Operating System for every process. The PCB is identified by an integer process ID (PID). A PCB keeps all the information needed to keep track of a process as listed below

Process State

The current state of the process i.e., whether it is ready, running, waiting, or whatever

Process privileges

This is required to allow/disallow access to system resources

Process ID
Unique identification for each of the process in the operating system

Pointer
A pointer to parent process

Program Counter
Program Counter is a pointer to the address of the next instruction to be executed for this process

CPU registers
Various CPU registers where process need to be stored for execution for running state

CPU Scheduling Information
Process priority and other scheduling information which is required to schedule the process

Memory management information
This includes the information of page table, memory limits, Segment table depending on memory used by the operating system

Accounting information
This includes the amount of CPU used for process execution, time limits, execution ID etc

IO status information
This includes a list of I/O devices allocated to the process

Process Scheduling

The process scheduling is the activity of the process manager that handles the removal of the running process from the CPU and the selection of another process on the basis of a particular strategy.

Process scheduling is an essential part of a Multiprogramming operating systems. Such operating systems allow more than one process to be loaded into the executable memory at a time and the loaded process shares the CPU using time multiplexing.

Process Scheduling Queues

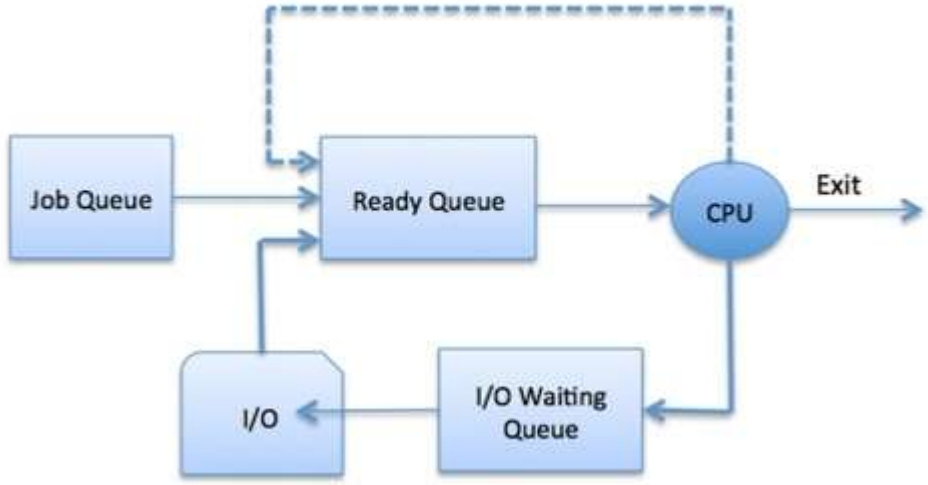
The OS maintains all PCBs in Process Scheduling Queues. The OS maintains a separate queue for each of the process states and PCBs of all processes in the same execution state are placed in the same queue. When the state of a process is changed, its PCB is unlinked from its current queue and moved to its new state queue.

The Operating System maintains the following important process scheduling queues -

Job queue − This queue keeps all the processes in the system

Ready queue − This queue keeps a set of all processes residing in main memory, ready and waiting to execute. A new process is always put in this queue

Device queues − The processes which are blocked due to unavailability of an I/O device constitute this queue



The OS can use different policies to manage each queue (FIFO, Round Robin, Priority, etc.). The OS scheduler determines how to move processes between the ready and run queues which can only have one entry per processor core on the system; in the above diagram, it has been merged with the CPU.

Operating System scheduling algorithms

A Process Scheduler schedules different processes to be assigned to the CPU based on particular scheduling algorithms. There are six popular process scheduling algorithms which we are going to discuss in this chapter –

- First-Come, First-Served (FCFS) Scheduling
- Shortest-Job-Next (SJN) Scheduling
- Priority Scheduling
- Shortest Remaining Time
- Round Robin(RR) Scheduling
- Multiple-Level Queues Scheduling

These algorithms are either **non-preemptive or preemptive**. Non-preemptive algorithms are designed so that once a process enters the running state, it cannot be preempted until it completes its allotted time, whereas the preemptive scheduling is based on priority where a scheduler may preempt a low priority running process anytime when a high priority process enters into a ready state.

First Come First Serve (FCFS)

- Jobs are executed on first come, first serve basis.
- It is a non-preemptive, pre-emptive scheduling algorithm.
- Easy to understand and implement.
- Its implementation is based on FIFO queue.
- Poor in performance as average wait time is high.

Wait time of each process is as follows -

Wait Time: Service Time - Arrival Time

Process

P0 0 – 0 = 0

P1 5 – 1 = 4

P2 8 - 2 = 6
P3 16 - 3 = 13

Average Wait Time: (0+4+6+13) / 4 = 5.75

Shortest Job Next (SJN)

- This is also known as **shortest job first**, or SJF
- This is a non-preemptive, pre-emptive scheduling algorithm.
- Best approach to minimize waiting time.
- Easy to implement in Batch systems where required CPU time is known in advance.
- Impossible to implement in interactive systems where required CPU time is not known.
- The processer should know in advance how much time process will take.

Given: Table of processes, and their Arrival time, Execution time

Process	Arrival Time	Execution Time	Service Time
P0	0	5	0
P1	1	3	5
P2	2	8	14
P3	3	6	8

Waiting time of each process is as follows –

Process Waiting Time

P0 0 - 0 = 0

P1 5 - 1 = 4

P2 14 - 2 = 12

P3 8 - 3 = 5

Average Wait Time: (0 + 4 + 12 + 5)/4 = 21 / 4 = 5.25

Priority Based Scheduling

- Priority scheduling is a non-preemptive algorithm and one of the most common scheduling algorithms in batch systems.
- Each process is assigned a priority. Process with highest priority is to be executed first and so on.
- Processes with same priority are executed on first come first served basis.
- Priority can be decided based on memory requirements, time requirements or any other resource requirement

Given:

Table of processes, and their Arrival time, Execution time, and priority. Here we are considering 1 is the lowest priority.

Process	Arrival Time	Execution Time	Priority	Service Time
P0	0	5	1	0
P1	1	3	2	11
P2	2	8	1	14
P3	3	6	3	5

Waiting time of each process is as follows −

Process	Waiting Time
P0	0 - 0 = 0
P1	11 - 1 = 10
P2	14 - 2 = 12
P3	5 - 3 = 2

Average Wait Time: (0 + 10 + 12 + 2)/4 = 24 / 4 = 6

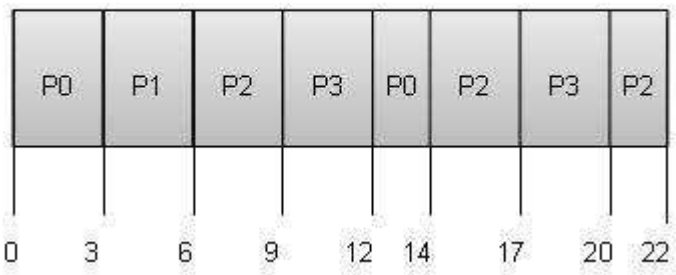
Shortest Remaining Time

- Shortest remaining time (SRT) is the preemptive version of the SJN algorithm.
- The processor is allocated to the job closest to completion but it can be preempted by a newer ready job with shorter time to completion.
- Impossible to implement in interactive systems where required CPU time is not known.
- It is often used in batch environments where short jobs need to give preference.

Round Robin Scheduling

- Round Robin is the preemptive process scheduling algorithm.
- Each process is provided a fix time to execute, it is called a **quantum**.
- Once a process is executed for a given time period, it is preempted and other process executes for a given time period.
- Context switching is used to save states of preempted processes.

Quantum = 3



Wait time of each process is as follows −

Process	Wait Time : Service Time - Arrival Time
P0	(0 - 0) + (12 - 3) = 9
P1	(3 - 1) = 2
P2	(6 - 2) + (14 - 9) + (20 - 17) = 12
P3	(9 - 3) + (17 - 12) = 11

Average Wait Time: (9+2+12+11) / 4 = 8.5

Multiple-Level Queues Scheduling

Multiple-level queues are not an independent scheduling algorithm. They make use of other existing algorithms to group and schedule jobs with common characteristics.

- Multiple queues are maintained for processes with common characteristics.
- Each queue can have its own scheduling algorithms.
- Priorities are assigned to each queue.

For example, CPU-bound jobs can be scheduled in one queue and all I/O-bound jobs in another queue. The Process Scheduler then alternately selects jobs from each queue and assigns them to the CPU based on the algorithm assigned to the queue.

Multiple-Processor Scheduling in Operating System

In multiple-processor scheduling **multiple CPU's** are available and hence **Load Sharing** becomes possible. However multiple processor scheduling is more **complex** as compared to single processor scheduling. In multiple processor scheduling there are cases when the processors are identical i.e. HOMOGENEOUS, in terms of their functionality, we can use any processor available to run any process in the queue.

Approaches to Multiple-Processor Scheduling –

One approach is when all the scheduling decisions and I/O processing are handled by a single processor which is called the **Master Server** and the other processors executes only the **user code**. This is simple and reduces the need of data sharing. This entire scenario is called **Asymmetric Multiprocessing**.

A second approach uses **Symmetric Multiprocessing** where each processor is **self-scheduling**. All processes may be in a common ready queue or each processor may have its own private queue for ready processes. The scheduling proceeds further by having the scheduler for each processor examine the ready queue and select a process to execute.

Processor Affinity

Processor Affinity means a processes has an **affinity** for the processor on which it is currently running.

When a process runs on a specific processor there are certain effects on the cache memory. The data most recently accessed by the process populate the cache for the processor and as a result successive memory access by the process are often satisfied in the cache memory. Now if the process migrates to another processor, the contents of the cache memory must be invalidated for the first processor and the cache for the second processor must be repopulated. Because of the high cost of invalidating and repopulating caches, most of the SMP(symmetric multiprocessing) systems try to avoid migration of processes from one processor to another and try to keep a process running on the same processor.

Processor Affinity:

There are two types of processor affinity:

Soft Affinity:

When an operating system has a policy of attempting to keep a process running on the same processor but not guaranteeing it will do so, this situation is called soft affinity.

Hard Affinity:

Hard Affinity allows a process to specify a subset of processors on which it may run. Some systems such as Linux implements soft affinity but also provide some system calls like *sched_setaffinity()* that supports hard affinity.

Load Balancing

Load Balancing is the **phenomena** which keeps the **workload** evenly **distributed** across all processors in an SMP system. Load balancing is necessary only on systems where each processor has its own private queue of process which are eligible to execute. Load balancing is unnecessary because once a processor becomes idle it immediately extracts a runnable process from the common run queue. On SMP(symmetric multiprocessing), it is important to keep the workload balanced among all processors to fully utilize the benefits of having more than one processor else one or more processor will sit idle while other processors have high workloads along with lists of processors awaiting the CPU.

There are two general approaches to load balancing:

Push Migration:

In push migration a task routinely checks the load on each processor and if it finds an imbalance then it evenly distributes load on each processors by moving the processes from overloaded to idle or less busy processors.

Pull Migration:

Pull Migration occurs when an idle processor pulls a waiting task from a busy processor for its execution.

Inter Process Communication

Inter Process Communication (IPC) refers to a mechanism, where the operating systems allow various processes to communicate with each other. This involves synchronizing their actions and managing shared data.

There are 2 types of process:

- Independent Processes – Processes which do not share data with other processes.
- Cooperating Processes – Processes that shares data with other processes.

Cooperating process require Inter process communication (IPC) mechanism.

Inter Process Communication is the mechanism by which cooperating process share data and information.

There are 2 ways by which Interprocess communication is achieved:

- Shared memory
- Message Parsing

Shared Memory

1. A particular region of memory is shared between cooperating process.
2. Cooperating process can exchange information by reading and writing data to this shared region.
3. It's faster than Memory Parsing, as Kernel is required only once, that is, setting up a shared memory. After That, kernel assistance is not required.

Message Parsing

1. Communication takes place by exchanging messages directly between cooperating process.
2. Easy to implement
3. Useful for small amount of data.

4. Implemented using System Calls, so takes more time than Shared Memory.

Process Synchronization

When several threads (or processes) share data, running in parallel on different cores, then changes made by one process may override changes made by another process running parallel. Resulting in inconsistent data. So, this requires processes to be synchronized, handling system resources and processes to avoid such situation is known as Process Synchronization.

Classic Banking Example:

- Consider your bank account has 5000$.
- You try to withdraw 4000$ using net banking and simultaneously try to withdraw via ATM too.
- For Net Banking at time t = 0ms bank checks you have 5000$ as balance and you're trying to withdraw 4000$ which is lesser than your available balance. So, it lets you proceed further and at time t = 1ms it connects you to server to transfer the amount
- Imagine, for ATM at time t = 0.5ms bank checks your available balance which currently is 5000$ and thus let's you enter ATM password and withdraw amount.
- At time t = 1.5 ms ATM dispenses the cash of 4000$ and at time t = 2 net banking transfer is complete of 4000$.

Effect on the system

Now, due to concurrent access and processing time that computer takes in both ways you were able to withdraw 3000$ more than your balance. In total 8000$ were taken out and balance was just 5000$.

How to solve this Situation

To avoid such situations **process synchronisation is used**, so another concurrent process P2 is notified of existing concurrent process P1 and not allowed to go through as there is P1 process which is running and P2 execution is only allowed once P1 completes.

Process Synchronization also prevents **race around condition**. It's the condition in which several processes access and manipulate the same data. In this condition, the outcome of the execution depends upon the particular order in which access takes place.

There are two ways any process can execute:

In Concurrent Execution:

The CPU scheduler switches rapidly between processes. A process is stopped at any points and the processor is assigned to another instruction execution. Here, only one instruction is executed at a time.

Parallel execution:

2 or more instructions of different process execute simultaneously on different processing cores.

Critical Section Problem:

Critical Section problem is a classic computer science problem in many banking systems, where there are many shared resources like memory, I/O etc. Concurrent access may override changes made by other process running in parallel.

The Critical Section Problem is to design a protocol which processes use to cooperate .To enter into critical section, a process requests permission through entry section code. After critical section code is executed, then there is exit section code, to indicate the same.

The main blocks of process are –

1. **Entry Section –** To enter the critical section code, a process must request permission. Entry Section code implements this request.
2. **Critical Section –** This is the segment of code where process changes common variables, updates a table, writes to a file and so on. **When 1 process is executing in its critical section, no other process is allowed to execute in its critical section.**
3. **Exit Section –** After the critical section is executed, this is followed by exit section code which marks the end of critical section code.
4. **Remainder Section –** The remaining code of the process is known as remaining section.

Solution for Process Synchronization:

To further solve such situations many more solutions are available which will be discussed in detail in further posts –

1. **Semaphores**
2. **Critical Section**
3. **Test and Set**
4. **Peterson's Solution**
5. **Mutex**

Mutex

Mutex lock is essentially a variable that is binary nature that provides code wise functionality for mutual exclusion. At times, there may be multiple threads that may be trying to access same resource like memory or I/O etc. To make sure that there is no overriding. Mutex provides a locking mechanism.

Only one thread at a time can take the ownership of a mutex and apply the lock. Once it done utilising the resource and it may release the mutex lock.

Semaphore in Operating System

Semaphore is also an entity devised by Edsger W. Dijkstra, to solve Process Synchronization problem in OS. Its most popular use is it solve Critical Section algorithm.

It uses signaling mechanism to allow access to shared resource namely by two –

1. Wait
2. Signal

Semaphore Types

There are two types of Semaphores –

1. Binary Semaphore – Only True/False or 0/1 values
2. Counting Semaphore – Non-negative value

Semaphore Implementation

Semaphore can have two different operations which are wait and signal. In some books wait signals are also denoted by P(s) and signal by V(s). Where s is a common semaphore.
Wait p(s) or wait(s)

1. Wait decrements the value of semaphore by 1

Is c Signal v(s) or signal(s)

1. Signal increments the value of semaphore by 1

Semaphore characteristics

1. Semaphore can only have non negative values
2. Before start of program it is always initialised to 1

For Binary Semaphore

Let us try to understand the above code with an example –

1. Imagine that there are two processes A and B.
2. At the beginning the value of semaphore is initialised as 1.
3. Imagine process A wants to enter the critical section
    1. Before it can do that it checks the value of semaphore which is 1 thus, it can enter the CS and semaphore value is turned to 0
4. Now imagine that process B wants to enter too
    1. It checks the semaphore value which is 0 thus it can't enter and waits until the value is non zero – non negative value
5. Now, Process A finishes and signals semaphore which in turns changes semaphore value to 1
6. Thus, now process B can enter Critical section

In this way mutual exclusion was achieved.

For Counting Semaphore

For Counting Semaphore we initialize the value of semaphore as the number of concurrent access of critical sections we want to allow.

For example Let us assume that the value is 3.

- Process 1 enters Critical section and semaphore value is changed to 2
- Process 2 also enters critical section and semaphore value is changed to 1
- Process 2 signals semaphore and comes out of critical section and Semaphore value is 2
- Note at this moment only 1 process that is process 1 is in critical section
- Process 3 and 4 also enter critical section simultaneously and semaphore value is 0
- At this moment there are three processes in Critical section which are process Process 1, 3, 4
- Now imagine that process 5 wants to enter the CS. It would not be able to enter as semaphore value is 0
- It can only enter once any of the process 1, 3, 4 signals out of the critical section.

Concurrent Atomic Transactions

The mutual exclusion of critical sections ensures that the critical sections are executed atomically. That is, if two critical sections are executed concurrently, the result is equivalent to their sequential execution in some unknown order. Although this property is useful in many application domains, in many cases we would like to make sure that a critical section forms a single logical unit of work that either is performed in its entirety or is not performed at all.

An example is funds transfer, in which one account is debited and another is credited. Clearly, it is essential for data consistency either that both the credit and debit occur or that neither occur. Consistency of data, along with storage and retrieval of data, is a concern often

associated with database systems. Recently, there has been an upsurge of interest in using database-systems techniques in operating systems.

Operating systems can be viewed as manipulators of data; as such, they can benefit from the advanced techniques and models available from database research. For instance, many of the ad hoc techniques used in operating systems to manage files could be more flexible and powerful if more formal database methods were used in their place. In Sections 6.9.2 to 6.9.4, we describe some of these database techniques and explain how they can be used by operating systems. First, however, we deal with the general issue of transaction atomicity. It is this property that the database techniques are meant to address.

System Model

A collection of instructions (or operations) that performs a single logical function is called a transaction. A major issue in processing transactions is the preservation of atomicity despite the possibility of failures within the computer system. We can think of a transaction as a program unit that accesses and perhaps updates various data items that reside on a disk within some files. From our point of view, such a transaction is simply a sequence of read and write operations terminated by either a commit operation or an abort operation.

A commit operation signifies that the transaction has terminated its execution successfully, whereas an abort operation signifies that the transaction has ended its normal execution due to some logical error or a system failure. If a terminated transaction has completed its execution successfully, it is committed; otherwise, it is aborted. Since an aborted transaction may already have modified the data that it has accessed, the state of these data may not be the same as it would have been if the transaction had executed atomically. So that atomicity is ensured an aborted transaction must have no effect on the state of the data that it has already modified. Thus, the state of the data accessed by an aborted transaction must be restored to what it was just before the transaction started executing.

We say that such a transaction has been rolled back. It is part of the responsibility of the system to ensure this property. To determine how the system should ensure atomicity, we need first to identify the properties of devices used for storing the various data accessed by the transactions. Various types of storage media are distinguished by their relative speed, capacity, and resilience to failure.

Volatile storage:

Information residing in volatile storage does not usually survive system crashes. Examples of such storage are main and cache memory. Access to volatile storage is extremely fast, both because of the speed of the memory access itself and because it is possible to access directly any data item in volatile storage.

Nonvolatile storage:

Information residing in nonvolatile storage usually survives system crashes. Examples of media for such storage are disks and magnetic tapes. Disks are more reliable than main memory but less reliable than magnetic tapes. Both disks and tapes, however, are subject to failure, which may result in loss of information. Currently, nonvolatile storage is slower than volatile storage by several orders of magnitude, because disk and tape devices are electromechanical and require physical motion to access data.

Stable storage:

Information residing in stable storage is never lost (never should be taken with a grain of salt, since theoretically such absolutes cannot be guaranteed). To implement an approximation of such storage, we need to replicate information in several nonvolatile storage caches (usually disk) with independent failure modes and to update the information in a controlled manner (Section 12.8). Here, we are concerned only with ensuring transaction atomicity in an environment where failures result in the loss of information on volatile storage.

Log-Based Recovery

One way to ensure atomicity is to record, on stable storage, information describing all the modifications made by the transaction to the various data it accesses. The most widely used method for achieving this form of recording is write-ahead logging. Here, the system maintains, on stable storage, a data structure called the log. Each log record describes a single operation of a transaction write and has the following fields:

- Transaction name. The unique name of the transaction that performed the write operation
- Data item name. The unique name of the data item written
- Old value. The value of the data item prior to the write operation
- New value.

In cases where the data are extremely important and fast failure recovery is necessary, the price is worth the functionality. Using the log, the system can handle any failure that does not result in the loss of information on nonvolatile storage. The recovery algorithm uses two procedures:

- undo(TJ), which restores the value of all data updated by transaction T, to the old values
- redo(Tj), which sets the value of all data updated by transaction T;

If a system failure occurs, we restore the state of all updated data by consulting the log to determine which transactions need to be redone and which need to be undone. This classification of transactions is accomplished as follows:

Checkpoints

When a system failure occurs, we must consult the log to determine those transactions that need to be redone and those that need to be undone. In principle, we need to search the entire log to make these determinations. To reduce these types of overhead, we introduce the concept of checkpoints. During execution, the system maintains the write-ahead log. In addition, the system periodically performs checkpoints that require the following sequence of actions to take place:

1. Output all log records currently residing in volatile storage (usually main memory) onto stable storage.
2. Output all modified data residing in volatile storage to the stable storage.
3. Output a log record onto stable storage. The presence of a record in the log allows the system to streamline its recovery procedure. Consider a transaction Tj that committed prior to the checkpoint.

Serializability

Consider a system with two data items, A and B, that are both read and written by two transactions, To and T\. Suppose that these transactions are executed atomically in the order To followed by T\. This execution sequence, which is called a schedule, is represented in Figure 6.22. In schedule 1 of Figure 6.22, the sequence of instruction steps is in chronological order from top to bottom, with instructions of To appearing in the left column and instructions of T\ appearing in the right column. A schedule in which each transaction is executed atomically is called a serial schedule. A serial schedule consists of a sequence of instructions from various transactions wherein the instructions belonging to a particular transaction appear together.

Thus, for a set of n transactions, there exist n\ different valid serial schedules. Each serial schedule is correct, because it is equivalent to the atomic execution of the various participating transactions in some arbitrary order. If we allow the two transactions to overlap their execution, then the resulting schedule is no longer serial. A nonserial schedule does not necessarily imply an incorrect execution (that is, an execution that is not equivalent to one represented by a serial schedule).

T_0	T_1
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Figure 6.22 Schedule 1: A serial schedule in which T_0 is followed by T_1.

read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Figure 6.23 Schedule 2: A concurrent serializable schedule.

To see that this is the case, we need to define the notion of conflicting operations. Consider a schedule S in which there are two consecutive operations O,- and Oj of transactions T, and Tj, respectively. We say that O, and Oj conflict if they access the same data item and at least one of them is a write operation. To illustrate the concept of conflicting operations, we consider the nonserial schedule 2 of Figure 6.23. The write(A) operation of To conflicts with the read(A) operation of Ti.

However, the write(A) operation of T\ does not conflict with the read(B) operation of To, because the two operations access different data items. Let Oj and Oj be consecutive operations of a schedule S. If O, and O; are operations of different transactions and O-, and Oj do not conflict, then we can swap the order of O, and 0/ to produce a new schedule S'. We expect S to be equivalent to S', as all operations appear in the same order in both schedules, except for O, and Oj, whose order does not matter. We can illustrate the swapping idea by considering again schedule 2 of Figure 6.23.

As the write(A) operation of T\ does not conflict with the read(B) operation of To, we can swap these operations to generate an equivalent schedule. Regardless of the initial system state, both schedules produce the same final system state. Continuing with this procedure of swapping non conflicting operations, we get:

- Swap the read(B) operation of TQ with the read(A) operation of T\.
- Swap the write(B) operation of To with the write(A) operation of T\.
- Swap the write(B) operation of To with the read(A) operation of T\.

The final result of these swaps is schedule 1 in Figure 6.22, which is a serial schedule. Thus, we have shown that schedule 2 is equivalent to a serial schedule. This result implies that, regardless of the initial system state, schedule 2 will produce the same final state as will some serial schedule. If a schedule S can be transformed into a serial schedule S' by a series of swaps of nonconflicting operations, we say that a schedule S is conflict serializable. Thus, schedule 2 is conflict serializable, because it can be transformed into **Locking Protocol**

Lock-Based Protocol

In this type of protocol, any transaction cannot read or write data until it acquires an appropriate lock on it. There are two types of lock:

1. Shared lock:

- It is also known as a Read-only lock. In a shared lock, the data item can only read by the transaction.
- It can be shared between the transactions because when the transaction holds a lock, then it can't update the data on the data item.

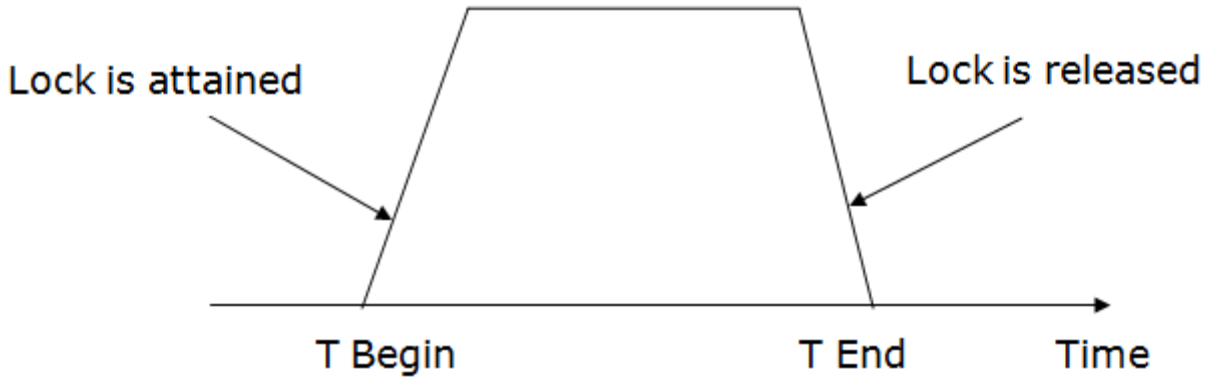
2. Exclusive lock:

- In the exclusive lock, the data item can be both reads as well as written by the transaction.
- This lock is exclusive, and in this lock, multiple transactions do not modify the same data simultaneously.

One protocol that ensures serializability is the two-phase locking protocol.

Two-phase locking (2PL)

- The two-phase locking protocol divides the execution phase of the transaction into three parts.
- In the first part, when the execution of the transaction starts, it seeks permission for the lock it requires.
- In the second part, the transaction acquires all the locks. The third phase is started as soon as the transaction releases its first lock.
- In the third phase, the transaction cannot demand any new locks. It only releases the acquired locks.



There are two phases of 2PL:

Growing phase: In the growing phase, a new lock on the data item may be acquired by the transaction, but none can be released.

Shrinking phase: In the shrinking phase, existing lock held by the transaction may be released, but no new locks can be acquired.
In the below example, if lock conversion is allowed then the following phase can happen:

1. Upgrading of lock (from S(a) to X (a)) is allowed in growing phase.
2. Downgrading of lock (from X(a) to S(a)) must be done in shrinking phase.

Timestamp-Based Protocols

In the locking protocols described above, the order followed by pairs of conflicting transactions is determined at execution time by the first lock that both request and that involves incompatible modes. Another method for determining the serializability order is to select an order in advance. The most common method for doing so is to use a timestamp ordering scheme. With each transaction T, in the system, we associate a unique fixed timestamp, denoted by TS(T/). This timestamp is assigned by the system 6.9 Atomic Transactions 229 before the transaction T, starts execution. If a transaction 7} has been assigned timestamp TS(Tj-), and later a new transaction 7) enters the system, then TS(7}) < TS(Tj). There are two simple methods for implementing this scheme:

Use the value of the system clock as the timestamp; that is, a transaction's timestamp is equal to the value of the clock when the transaction enters the system. This method will not work for transactions that occur on separate systems or for processors that do not share a clock.

Use a logical counter as the timestamp; that is, a transaction's timestamp is equal to the value of the counter when the transaction enters the system. The counter is incremented after a new timestamp is assigned. The timestamps of the transactions determine the serializability order. Thus, if TS(T,) < TS(T,), then the system must ensure that the produced schedule is equivalent to a serial schedule in which transaction T, appears before transaction T,. To implement this scheme, we associate with each data item Q two timestamp values:

W-timestamp(Q) denotes the largest timestamp of any transaction that successfully executed write(Q).

R-timestamp(Q) denotes the largest timestamp of any transaction that successfully executed read(Q). These timestamps are updated whenever a new read(Q) or write(Q) instruction is executed. The timestamp-ordering protocol ensures that any conflicting read and write operations are executed in timestamp order. This protocol operates as follows:

Suppose that transaction T,- issues read(Q): o If TS(T,) < W-timestamp(), then T, needs to read a value of Q that was already overwritten. Hence, the read operation is rejected, and Tj- is rolled back. o If TS(TJ) > W-timestamp(Q), then the read operation is executed, and R-timestamp(Q) is set to the maximum of R-timestamp(Q) and TS(T,).

Suppose that transaction 7} issues write(Q): o If TS(T,) < R-timestamp(Q), then the value of Q that 7} is producing was needed previously and T,- assumed that this value would never be produced. Hence, the write operation is rejected, and 7} is rolled back. => If TS(T,) < W-timestamp(Q), then T, is attempting to write an obsolete value of Q. Hence, this write operation is rejected, and T, is rolled back. o Otherwise, the write operation is executed. A transaction T, that is rolled back as a result of the issuing of either a read or write operation is assigned a new timestamp and is restarted.

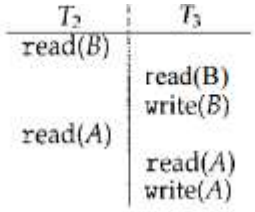


Figure 6.24 Schedule 3: A schedule possible under the timestamp protocol.

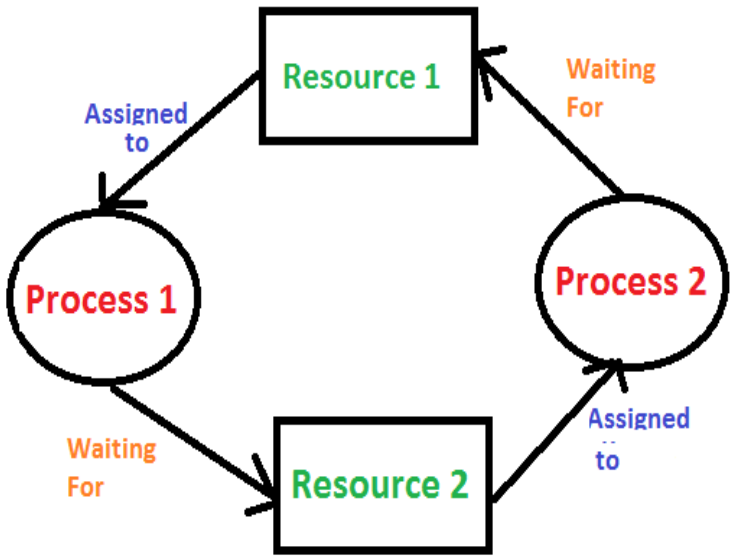
To illustrate this protocol, consider schedule 3 of Figure 6.24, which includes transactions % and T3. We assume that a transaction is assigned a timestamp immediately before its first instruction. Thus, in schedule 3, TS(T2) < TS(T3), and the schedule is possible under the timestamp protocol. This execution can also be produced by the two-phase locking protocol. However, some schedules are possible under the two-phase locking protocol but not under the timestamp protocol, and vice versa. The timestamp protocol ensures conflict serializability. This capability follows from the fact that conflicting operations are processed in timestamp order. The protocol also ensures freedom from deadlock, because no transaction ever waits.

UNIT - III
INTRODUCTION OF DEADLOCK IN OPERATING SYSTEM

Process in operating systems uses different resources and uses resources in following way.

1. Requests a resource
2. Use the resource
3. Releases the resource

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. Consider an example when two trains are coming toward each other on same track and there is only one track, none of the trains can move once they are in front of each other. Similar situation occurs in operating systems when there are two or more processes hold some resources and wait for resources held by other(s). 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.



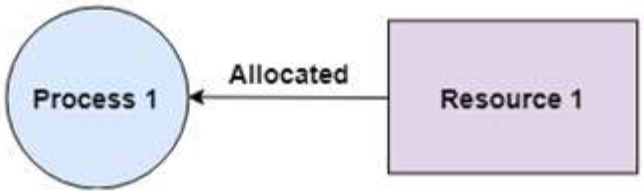
Deadlock Characterization

A deadlock happens in operating system when two or more processes need some resource to complete their execution that is held by the other process.

A deadlock occurs if the four Coffman conditions hold true. But these conditions are not mutually exclusive. They are given as follows:

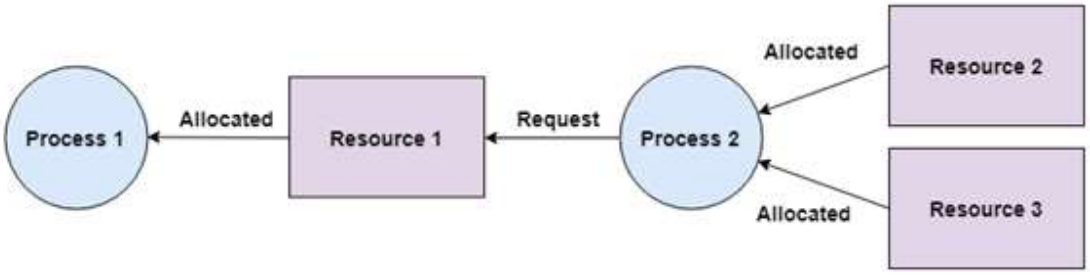
Mutual Exclusion

There should be a resource that can only be held by one process at a time. In the diagram below, there is a single instance of Resource 1 and it is held by Process 1 only.



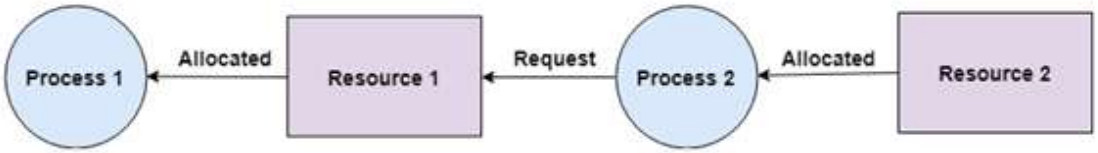
Hold and Wait

A process can hold multiple resources and still request more resources from other processes which are holding them. In the diagram given below, Process 2 holds Resource 2 and Resource 3 and is requesting the Resource 1 which is held by Process 1.



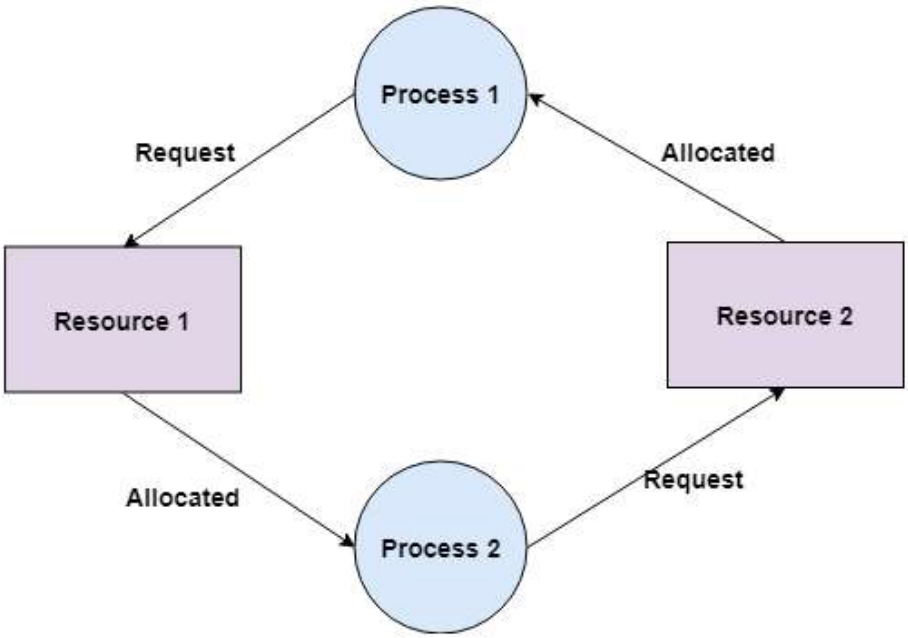
No Preemption

A resource cannot be preempted from a process by force. A process can only release a resource voluntarily. In the diagram below, Process 2 cannot preempt Resource 1 from Process 1. It will only be released when Process 1 relinquishes it voluntarily after its execution is complete.



Circular Wait

A process is waiting for the resource held by the second process, which is waiting for the resource held by the third process and so on, till the last process is waiting for a resource held by the first process. This forms a circular chain. For example: Process 1 is allocated Resource2 and it is requesting Resource 1. Similarly, Process 2 is allocated Resource 1 and it is requesting Resource 2. This forms a circular wait loop.



Deadlock Prevention

We can prevent Deadlock by eliminating any of the below four conditions.

Eliminate Mutual Exclusion

It is not possible to dis-satisfy the mutual exclusion because some resources, such as the tape drive and printer, are inherently non-shareable.

Eliminate Hold and wait

1. Allocate all required resources to the process before the start of its execution, this way hold and wait condition is eliminated but it will lead to low device utilization. for example, if a process requires printer at a later time and we have allocated printer before the start of its execution printer will remain blocked till it has completed its execution.
2. The process will make a new request for resources after releasing the current set of resources. This solution may lead to starvation.

Eliminate No Preemption

Preempt resources from the process when resources required by other high priority processes.

Eliminate Circular Wait

Each resource will be assigned with a numerical number. A process can request the resources increasing/decreasing. order of numbering.

For Example, if P1 process is allocated R5 resources, now next time if P1 ask for R4, R3 lesser than R5 such request will not be granted, only request for resources more than R5 will be granted.

Deadlock Avoidance

Deadlock avoidance can be done with Banker's Algorithm.

Banker's Algorithm

Bankers's Algorithm is resource allocation and deadlock avoidance algorithm which test all the request made by processes for resources, it checks for the safe state, if after granting request system remains in the safe state it allows the request and if there is no safe state it doesn't allow the request made by the process.

Following **Data structures** are used to implement the Banker's Algorithm:

Let **'n'** be the number of processes in the system and **'m'** be the number of resources types.

Available :

- It is a 1-d array of size **'m'** indicating the number of available resources of each type.
- Available[j] = k means there are **'k'** instances of resource type $\mathbf{R_j}$

Max :

- It is a 2-d array of size '**n*m'** that defines the maximum demand of each process in a system.
- Max[i, j] = k means process $\mathbf{P_i}$ may request at most **'k'** instances of resource type $\mathbf{R_j}$.

Allocation :

- It is a 2-d array of size **'n*m'** that defines the number of resources of each type currently allocated to each process.
- Allocation[i, j] = k means process $\mathbf{P_i}$ is currently allocated **'k'** instances of resource type $\mathbf{R_j}$

Need :

- It is a 2-d array of size **'n*m'** that indicates the remaining resource need of each process.
- Need [i, j] = k means process $\mathbf{P_i}$ currently need **'k'** instances of resource type $\mathbf{R_j}$
- for its execution.
- Need [i, j] = Max [i, j] – Allocation [i, j]

$Allocation_i$ specifies the resources currently allocated to process P_i and Need specifies the additional resources that process P_i may still request to complete its task.

Banker's algorithm consists of Safety algorithm and Resource request algorithm

Safety Algorithm

The algorithm for finding out whether or not a system is in a safe state can be described as follows:

1) Let Work and Finish be vectors of length 'm' and 'n' respectively.
Initialize: Work = Available
Finish[i] = false; for i=1, 2, 3, 4….n
2) Find an i such that both
a) Finish[i] = false
b) $Need_i$<= Work
if no such i exists goto step (4)
3) Work = Work + Allocation[i]
Finish[i] = true
goto step (2)
4) if Finish [i] = true for all i
then the system is in a safe state

Resource-Request Algorithm

Let $Request_i$ be the request array for process P_i. $Request_i[j] = k$ means process P_i wants k instances of resource type R_j. When a request for resources is made by process P_i, the following actions are taken:

1) If $Request_i$<= $Need_i$
Goto step (2) ; otherwise, raise an error condition, since the process has exceeded its maximum claim.

2) If $Request_i$<= Available
Goto step (3); otherwise, P_i must wait, since the resources are not available.

3) Have the system pretend to have allocated the requested resources to process Pi by modifying the state as
follows:
Available = Available – Requesti
$Allocation_i$ = $Allocation_i$ + $Request_i$
$Need_i$ = $Need_i$– $Request_i$

Resource Allocation Graph (RAG)

As Banker's algorithm using some kind of table like allocation, request, available all that thing to understand what is the state of the system. Similarly, if you want to understand the state of the system instead of using those table, actually tables are very easy to represent and understand it, but then still you could even represent the same information in the graph. That graph is called **Resource**

Allocation Graph (RAG).

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, but then you might not be able to know that by looking at the table. But the tables are better if the system contains lots of process and resource and Graph is better if the system contains less number of process and resource.

We know that any graph contains vertices and edges. So RAG also contains vertices and edges.

In RAG vertices are two type:

Process vertex - Every process will be represented as a process vertex. Generally, the process will be represented with a circle.

Resource vertex - Every resource will be represented as a resource vertex.

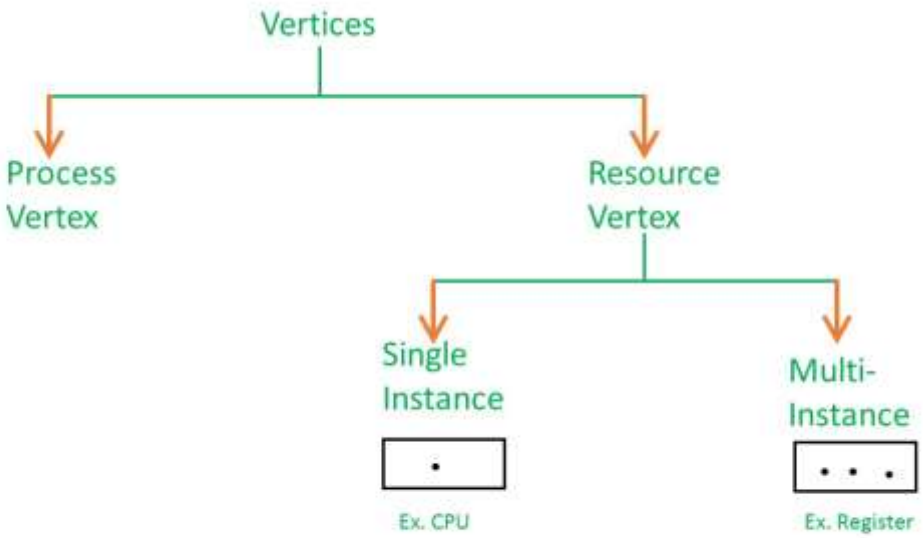
It is also two types:

Single instance type resource:

It represents as a box, inside the box, there will be one dot. So the number 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.

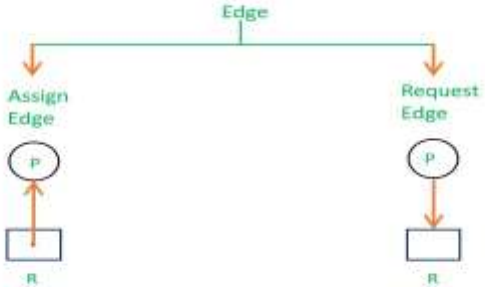


Now coming to the edges of RAG

There are two types of edges in RAG:

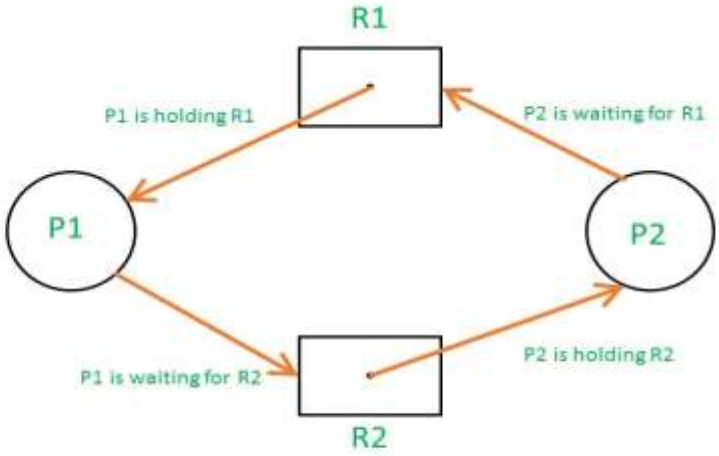
1. Assign Edge – If you already assign a resource to a process then it is called Assign edge.

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



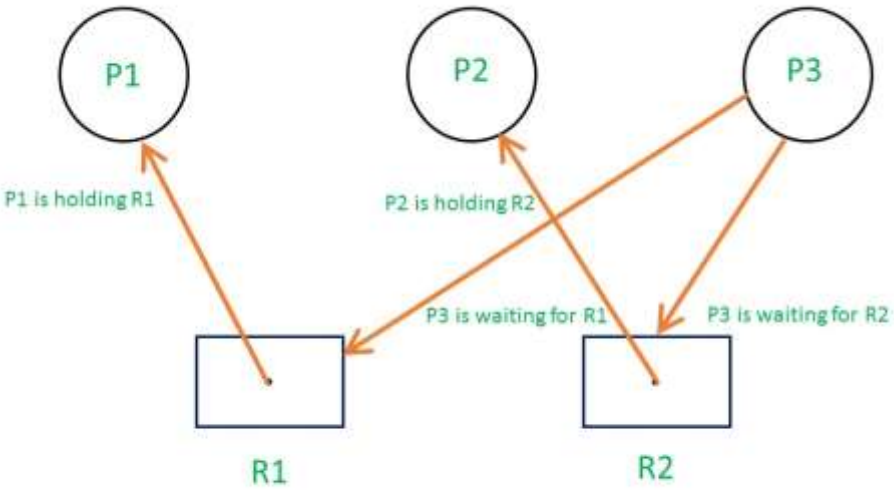
So, if a process is using a resource, an arrow is drawn from the resource node to the process node. If a process is requesting a resource, an arrow is drawn from the process node to the resource node.

Example 1 (Single instances RAG) –



SINGLE INSTANCE RESOURCE TYPE WITH DEADLOCK

If there is a cycle in the Resource Allocation Graph and each resource in the cycle provides only one instance, then the processes will be in deadlock. For example, if process P1 holds resource R1, process P2 holds resource R2 and process P1 is waiting for R2 and process P2 is waiting for R1, then process P1 and process P2 will be in deadlock.

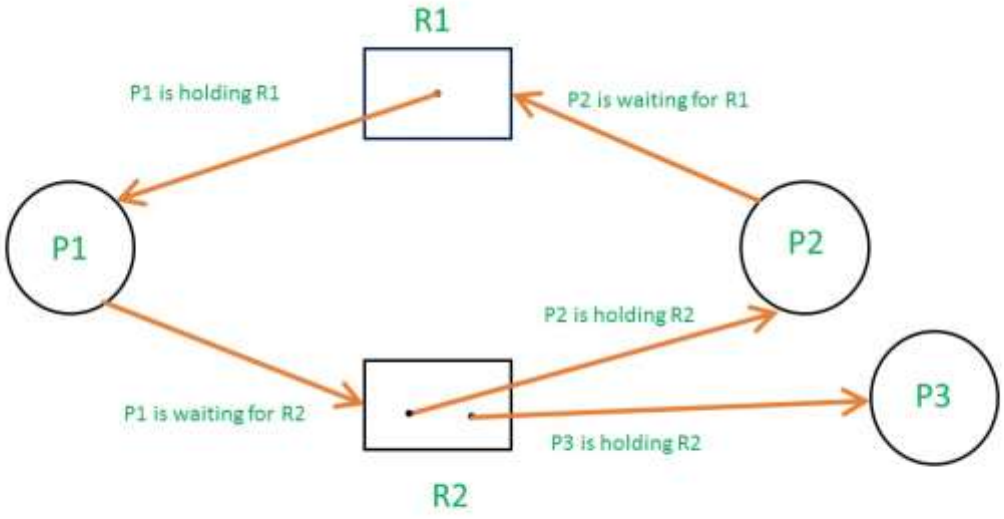


SINGLE INSTANCE RESOURCE TYPE WITHOUT DEADLOCK

Here's another example, that shows Processes P1 and P2 acquiring resources R1 and R2 while process P3 is waiting to acquire both resources. In this example, there is no deadlock because there is no circular dependency.

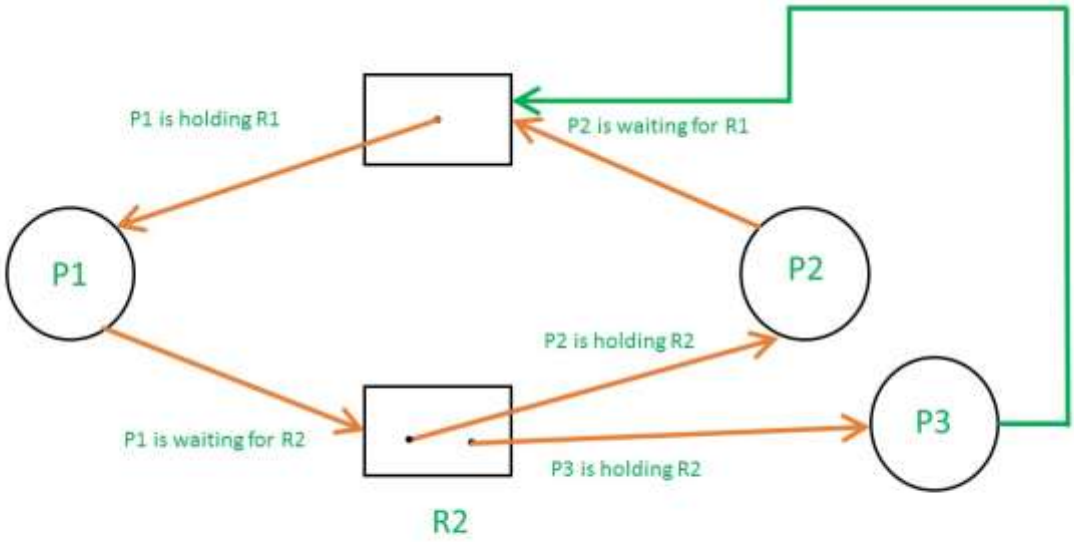
So cycle in single-instance resource type is the sufficient condition for deadlock.

Example 2 (Multi-instances RAG) –



MULTI INSTANCES WITHOUT DEADLOCK

From the above example, it is not possible to say the RAG is in a safe state or in an unsafe state.



MULTI INSTANCES WITH DEADLOCK

Recovery from Deadlock

When a Deadlock Detection Algorithm determines that a deadlock has occurred in the system, the system must recover from that deadlock. There are two approaches of breaking a Deadlock:

1. Process Termination:

To eliminate the deadlock, we can simply kill one or more processes. For this, we use two methods:

(a) Abort all the Deadlocked Processes:

Aborting all the processes will certainly break the deadlock, but with a great expenses. The deadlocked processes may have computed for a long time and the result of those partial computations must be discarded and there is a probability to recalculate them later.

(b)Abort one process at a time untill deadlock is eliminated:

Abort one deadlocked process at a time, untill deadlock cycle is eliminated from the system. Due to this method, there may be considerable overhead, because after aborting each process, we have to run deadlock detection algorithm to check whether any processes are still deadlocked.

Resource Preemption:

To eliminate deadlocks using resource preemption, we preepmt some resources from processes and give those resources to other processes.

This method will raise three issues:

(a) Selecting a victim:

We must determine which resources and which processes are to be preempted and also the order to minimize the cost.

(b) Rollback:

We must determine what should be done with the process from which resources are preempted. One simple idea is total rollback. That means abort the process and restart it.

(c) Starvation:

In a system, it may happen that same process is always picked as a victim. As a result, that process will never complete its designated task. This situation is called **Starvation** and must be avoided. One solution is that a process must be picked as a victim only a finite number of times.

UNIT - IV
MEMORY MANAGEMENT

Background:
Program must be brought into memory and placed within a process for it to be run"

Input queue or job queue:
Collection of processes on the disk that are waiting to be brought into memory to run the program"

User programs go through several steps before being run"

Binding of Instructions and Data to Memory!
Address binding of instructions and data to memory addresses can happen at three different stages"

Compile time:
If memory location known a priori, *absolute code* can be generated; must recompile code if starting location changes"

Load time: Must generate *relocatable code* if memory location is not known at compile time"

Execution time: Binding delayed until run time if the process can be moved during its execution from one memory segment to another. Need hardware support for address maps (e.g., *base* and *limit registers*).

Logical vs. Physical Address Space: The concept of a logical *address space* that is bound to a separate *physical address space* is central to proper memory management"

Logical address: Generated by the CPU; also referred to as "*virtual address*"

Physical address: Address seen by the memory unit"

Logical and physical addresses are the same in compile-time and load-time address-binding schemes; logical (virtual) and physical addresses differ in execution-time address-binding scheme"

Memory-Management Unit (MMU)
- Hardware device that maps virtual to physical address
- In MMU scheme, the value in the relocation register is added to every address generated by a user process at the time it is sent to memory
- The user program deals with *logical* addresses; it never sees the *Real* physical addresses"

Swapping:
A process can be swapped temporarily out of memory to a backing store, and then brought back into memory for continued execution

Backing store:
Fast disk large enough to accommodate copies of all memory images for all users; must provide direct access to these memory images

Roll out, roll in:

- Swapping variant used for priority-based scheduling algorithms; lower-priority process is swapped out so higher-priority process can be loaded and executed
- Major part of swap time is transfer time; total transfer time is directly proportional to the amount of memory swapped
- Modified versions of swapping are found on many systems (i.e.,UNIX, Linux, and Windows)

Demand Paging!

- Bring a page into memory only when it is needed
- Less I/O needed
- Less memory needed
- Faster response
- More users
- Page is needed ⇒ reference to it
- invalid reference ⇒ abort
- not-in-memory ⇒ bring to memory.

Memory Allocation

Main memory usually has two partitions:

- **Low Memory** − Operating system resides in this memory.
- **High Memory** − User processes are held in high memory.

Operating system uses the following memory allocation mechanism.

S.N.	Memory Allocation & Description
1	**Single-partition allocation** In this type of allocation, relocation-register scheme is used to protect user processes from each other, and from changing operating-system code and data. Relocation register contains value of smallest physical address whereas limit register contains range of logical addresses. Each logical address must be less than the limit register.
2	**Multiple-partition allocation** In this type of allocation, main memory is divided into a number of fixed-sized partitions where each partition should contain only one process. When a partition is free, a process is selected from the input queue and is loaded into the free partition. When the process terminates, the partition becomes available for another process.

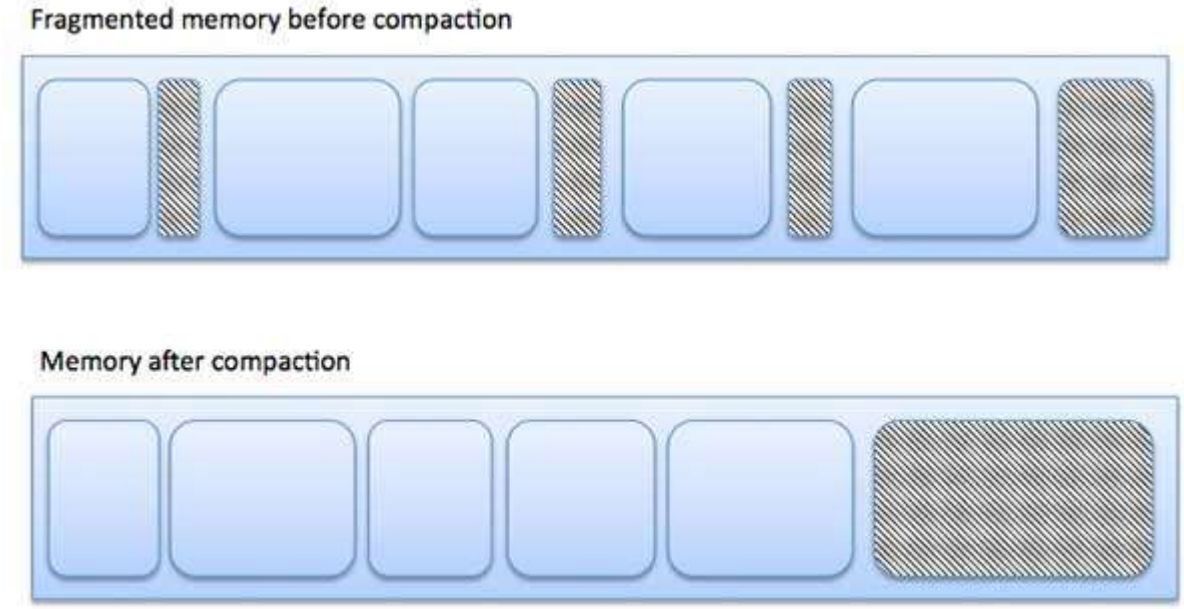
Fragmentation

As processes are loaded and removed from memory, the free memory space is broken into little pieces. It happens after sometimes that processes cannot be allocated to memory blocks considering their small size and memory blocks remains unused. This problem is known as Fragmentation.

Fragmentation is of two types:

Sr	Fragmentation & Description
1	**External fragmentation** Total memory space is enough to satisfy a request or to reside a process in it, but it is not contiguous, so it cannot be used.
2	**Internal fragmentation** Memory block assigned to process is bigger. Some portion of memory is left unused, as it cannot be used by another process.

The following diagram shows how fragmentation can cause waste of memory and a compaction technique can be used to create more free memory out of fragmented memory –



External fragmentation can be reduced by compaction or shuffle memory contents to place all free memory together in one large block. To make compaction feasible, relocation should be dynamic.

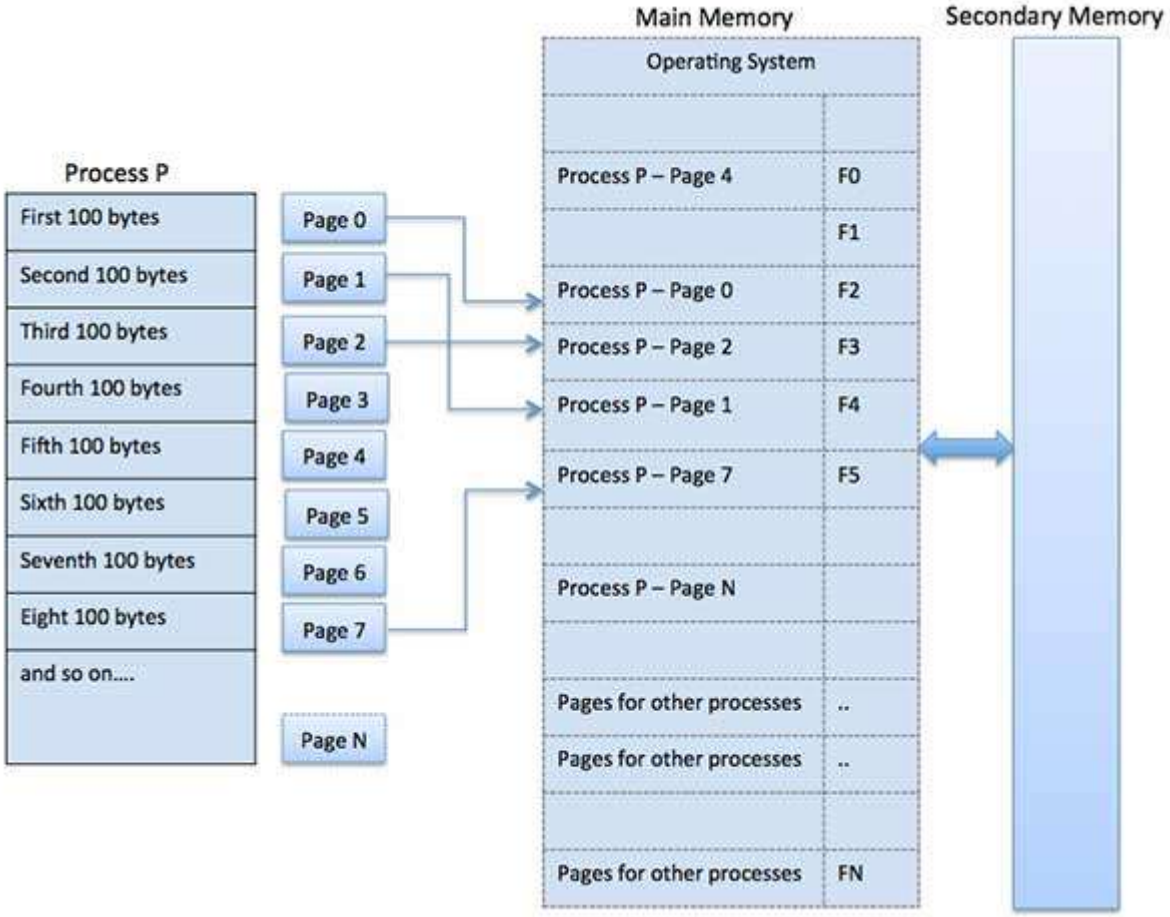
The internal fragmentation can be reduced by effectively assigning the smallest partition but large enough for the process.

Paging

A computer can address more memory than the amount physically installed on the system. This extra memory is actually called virtual memory and it is a section of a hard that's set up to emulate the computer's RAM. Paging technique plays an important role in implementing virtual memory.

Paging is a memory management technique in which process address space is broken into blocks of the same size called **pages** (size is power of 2, between 512 bytes and 8192 bytes). The size of the process is measured in the number of pages.

Similarly, main memory is divided into small fixed-sized blocks of (physical) memory called **frames** and the size of a frame is kept the same as that of a page to have optimum utilization of the main memory and to avoid external fragmentation.



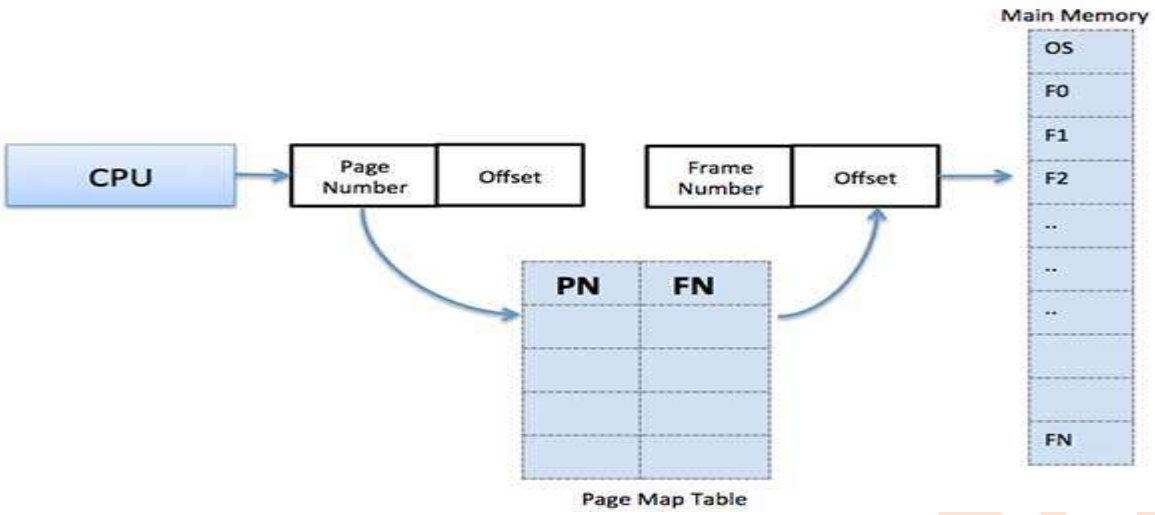
Address Translation

Page address is called **logical address** and represented by **page number** and the **offset**.
Logical Address = Page number + page offset

Frame address is called **physical address** and represented by a **frame number** and the **offset**.

Physical Address = Frame number + page offset

A data structure called **page map table** is used to keep track of the relation between a page of a process to a frame in physical memory.



When the system allocates a frame to any page, it translates this logical address into a physical address and create entry into the page table to be used throughout execution of the program.

When a process is to be executed, its corresponding pages are loaded into any available memory frames. Suppose there is a program of 8Kb but the memory can accommodate only 5Kb at a given point in time, then the paging concept will come into picture. When a computer runs out of RAM, the operating system (OS) will move idle or unwanted pages of memory to secondary memory to free up RAM for other processes and brings them back when needed by the program.

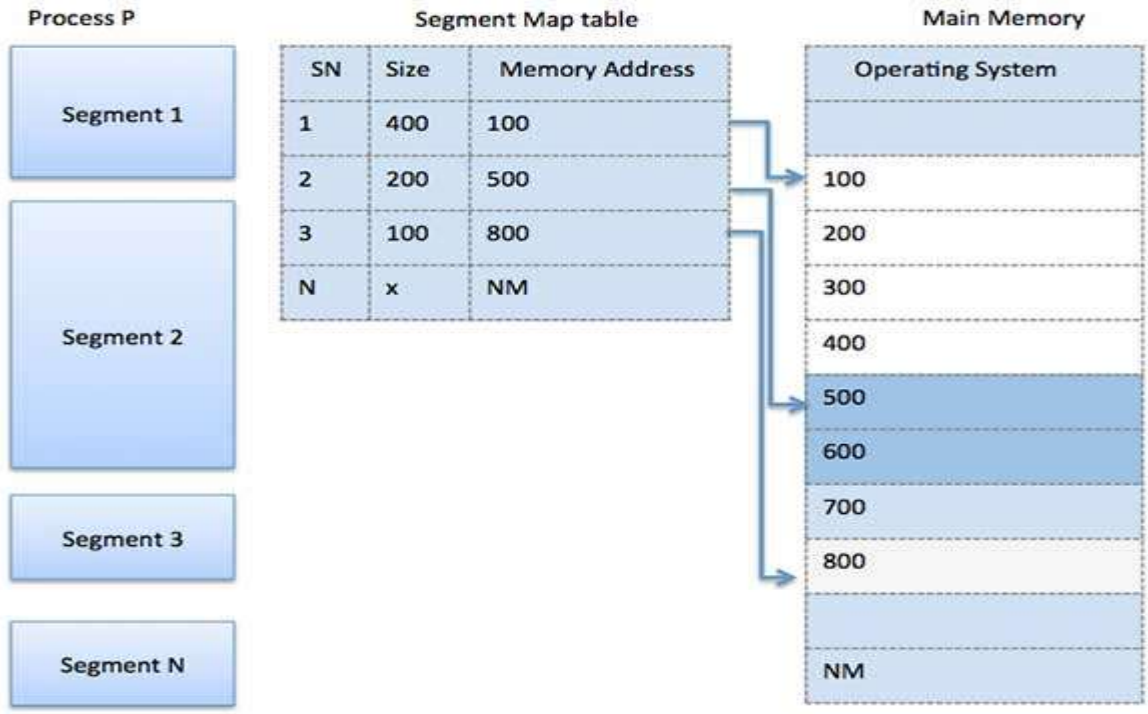
This process continues during the whole execution of the program where the OS keeps removing idle pages from the main memory and write them onto the secondary memory and bring them back when required by the program.

Segmentation

Segmentation is a memory management technique in which each job is divided into several segments of different sizes, one for each module that contains pieces that perform related functions. Each segment is actually a different logical address space of the program.

When a process is to be executed, its corresponding segmentation are loaded into non-contiguous memory though every segment is loaded into a contiguous block of available memory.
Segmentation memory management works very similar to paging but here segments are of variable-length where as in paging, pages are of fixed size.

A program segment contains the program's main function, utility functions, data structures, and so on. The operating system maintains a **segment map table** for every process and a list of free memory blocks along with segment numbers, their size and corresponding memory locations in main memory. For each segment, the table stores the starting address of the segment and the length of the segment. A reference to a memory location includes a value that identifies a segment and an offset.



Replacement Algorithms in Operating Systems

In an operating system that uses paging for memory management, a page replacement algorithm is needed to decide which page needs to be replaced when new page comes in.

Page Fault – A page fault happens when a running program accesses a memory page that is mapped into the virtual address space, but not loaded in physical memory.

Since actual physical memory is much smaller than virtual memory, page faults happen. In case of page fault, Operating System might have to replace one of the existing pages with the newly needed page. Different page replacement algorithms suggest different ways to decide which page to replace. The target for all algorithms is to reduce the number of page faults.

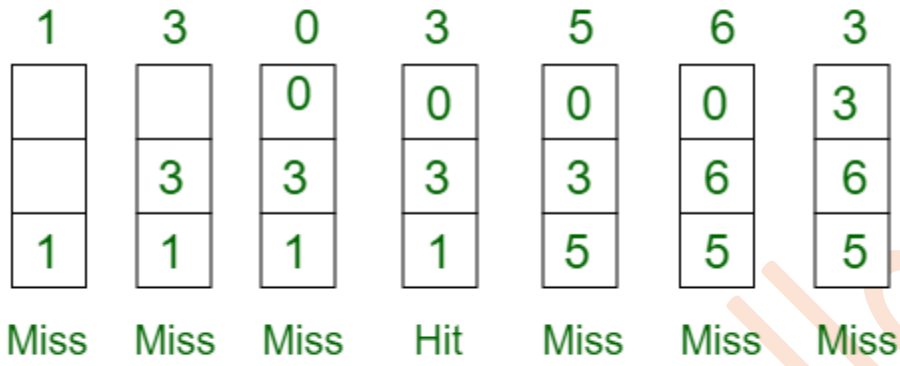
Page Replacement Algorithms:

First In First Out (FIFO) –

This is the simplest page replacement algorithm. In this algorithm, the operating system keeps track of all pages in the memory in a queue, the oldest page is in the front of the queue. When a page needs to be replaced page in the front of the queue is selected for removal.

Example-1 Consider page reference string 1, 3, 0, 3, 5, 6 with 3 page frames. Find the number of page faults.

Page reference 1, 3, 0, 3, 5, 6, 3



Total Page Fault = 6

Initially all slots are empty, so when 1, 3, 0 come they are allocated to the empty slots —> **3 Page Faults.**

when 3 comes, it is already in memory so —> **0 Page Faults.**
Then 5 comes, it is not available in memory so it replaces the oldest page slot i.e 1. —>**1 Page Fault.**

When 6 comes, it is also not available in memory so it replaces the oldest page slot i.e 3 —>**1 Page Fault.**

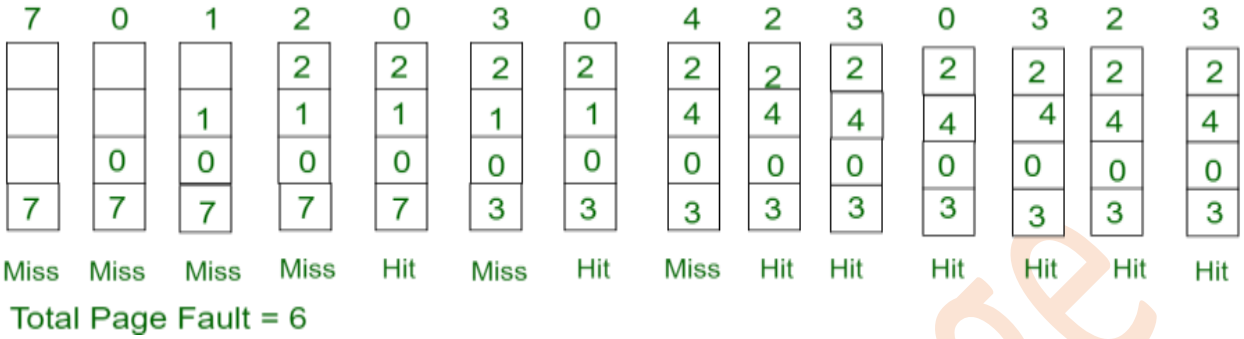
Finally when 3 comes it is not available so it replaces 0 **1 page fault**

Belady's anomaly:
Belady's anomaly proves that it is possible to have more page faults when increasing the number of page frames while using the First in First Out (FIFO) page replacement algorithm. For example, if reference string 3, 2, 1, 0, 3, 2, 4, 3, 2, 1, 0, 4 and 3 slots, we get 9 total page faults, but if the slots are increased to 4, the result is 10 page faults.

Optimal Page replacement –
In this algorithm, pages are replaced which would not be used for the longest duration of time in the future.

Example-2: Consider the page references 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, with 4 page frame. Find number of page fault.

Page reference 7,0,1,2,0,3,0,4,2,3,0,3,2,3 No. of Page frame - 4



Total Page Fault = 6

Initially all slots are empty, so when 7 0 1 2 are allocated to the empty slots —> **4 Page faults**
0 is already there so —> **0 Page fault.**
when 3 comes it will take the place of 7 because it is not used for the longest duration of time in the future.—>**1 Page fault.**
0 is already there so —> **0 Page fault.**.
4 will takes place of 1 —> **1 Page Fault.**

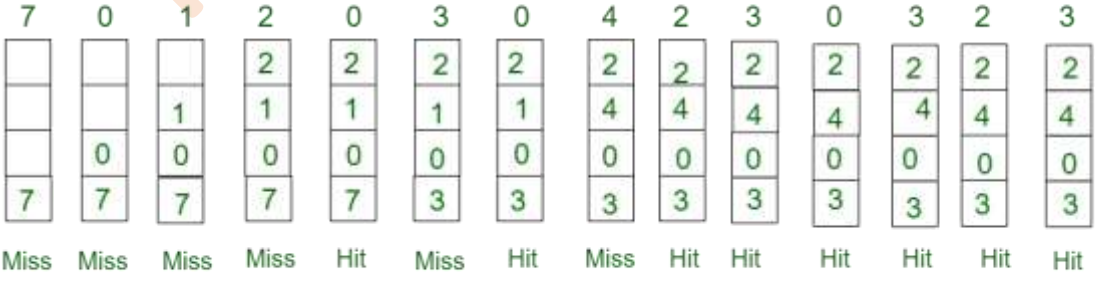
Now for the further page reference string —> **0 Page fault** because they are already available in the memory.
Optimal page replacement is perfect, but not possible in practice as the operating system cannot know future requests. The use of Optimal Page replacement is to set up a benchmark so that other replacement algorithms can be analyzed against it.

Least Recently Used:

In this algorithm page will be replaced which is least recently used.

Example-3 Consider the page reference string 7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2 with 4 page frames. Find number of page faults.

Page reference 7,0,1,2,0,3,0,4,2,3,0,3,2,3 No. of Page frame - 4



Total Page Fault = 6

Here LRU has same number of page fault as optimal but it may differ according to question.

Initially all slots are empty, so when 7 0 1 2 are allocated to the empty slots —> **4 Page faults**
0 is already there so —> **0 Page fault.**
when 3 come it will take the place of 7 because it is least recently used —>**1 Page fault**
0 is already in memory so —> **0 Page fault**.
4 will takes place of 1 —> **1 Page Fault**
Now for the further page reference string —> **0 Page fault** because they are already available in the memory.

UNIT - V
FILE MANAGEMENT

File Access Methods in Operating System

When a file is used, information is read and accessed into computer memory and there are several ways to access this information of the file. 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.

There are three ways to access a file into a computer system: Sequential-Access, Direct Access, Index sequential Method.

Sequential Access

It is the simplest access method. Information in the file is processed in order, one record after the other. This mode of access is by far the most common; for example, editor and compiler usually access the file in this fashion.

Read and write make up the bulk of the operation on a file. A read operation -*read next*- read the next position of the file and automatically advance a file pointer, which keeps track I/O location. Similarly, for the write *write next* append to the end of the file and advance to the newly written material.

Key points:

- Data is accessed one record right after another record in an order.
- When we use read command, it move ahead pointer by one
- When we use write command, it will allocate memory and move the pointer to the end of the file
- Such a method is reasonable for tape.

Direct Access

Another method is *direct access method* also known as *relative access method*. A filed-length logical record that allows the program to read and write record rapidly. in no particular order. The direct access is based on the disk model of a file since disk allows random access to any file block. For direct access, the file is viewed as a numbered sequence of block or record. Thus, we may read block 14 then block 59 and then we can write block 17. There is no restriction on the order of reading and writing for a direct access file. A block number provided by the user to the operating system is normally a *relative block number*, the first relative block of the file is 0 and then 1 and so on

Index sequential method:

It is the other method of accessing a file which is built on the top of the direct access method. These methods construct an index for the file. The index, like an index in the back of a book, contains the pointer to the various blocks. To find a record in the file, first search the index and then by the help of pointer access the file directly.

File structure:

A File Structure needs to be predefined format in such a way that an operating system under stands . It has an exclusively defined structure, which is based on its type.

Three types of files structure in OS:

- A text file: It is a series of characters that is organized in lines.

- An object file: It is a series of bytes that is organized into blocks.
- A source file: It is a series of functions and processes.

File Attributes

A file has a name and data. Moreover, it also stores meta information like file creation date and time, current size, last modified date, etc. All this information is called the attributes of a file system.

Here, are some important File attributes used in OS:

- **Name:** It is the only information stored in a human-readable form.
- **Identifier**: Every file is identified by a unique tag number within a file system known as an identifier.
- **Location:** Points to file location on device.
- **Type:** This attribute is required for systems that support various types of files.
- **Size**. Attribute used to display the current file size.
- **Protection**. This attribute assigns and controls the access rights of reading, writing, and executing the file.
- **Time, date and security:** It is used for protection, security, and also used for monitoring

File Type

It refers to the ability of the operating system to differentiate various types of files like text files, binary, and source files. However, Operating systems like MS_DOS and UNIX has the following type of files:

Character Special File

It is a hardware file that reads or writes data character by character, like mouse, printer, and more.

Ordinary files

- These types of files stores user information.
- It may be text, executable programs, and databases.
- It allows the user to perform operations like add, delete, and modify.

Directory Files

Directory contains files and other related information about those files. It is basically a folder to hold and organize multiple files.

Special Files

These files are also called device files. It represents physical devices like printers, disks, networks, flash drive, etc.

Functions of File

- Create file, find space on disk, and make an entry in the directory.
- Write to file, requires positioning within the file
- Read from file involves positioning within the file
- Delete directory entry, regain disk space.
- Reposition: move read/write position.

Commonly used terms in File systems

Field:

This element stores a single value, which can be static or variable length.

Database:

Collection of related data is called a database. Relationships among elements of data are explicit.

FILES:

Files is the collection of similar record which is treated as a single entity.

Record:

A Record type is a complex data type that allows the programmer to create a new data type with the desired column structure. Its groups one or more columns to form a new data type. These columns will have their own names and data type.

Structures of Directory in Operating System

A **directory** is a container that is used to contain folders and file. It organizes files and folders into a hierarchical manner.

There are several logical structures of a directory, these are given below.

Single-level directory:

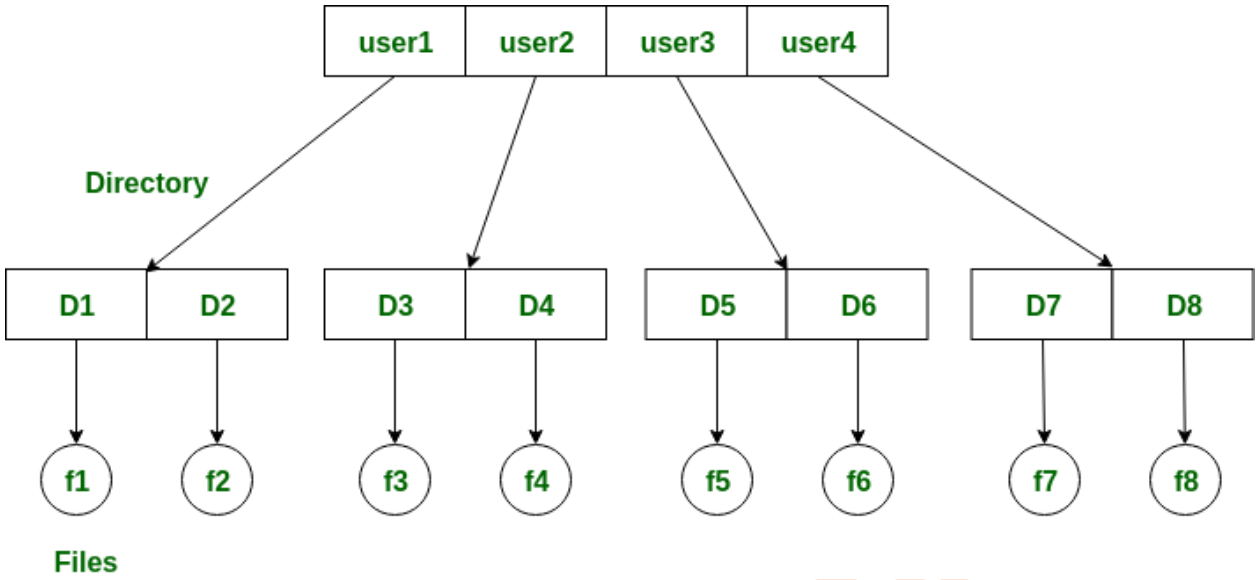
Single level directory is simplest directory structure.In it all files are contained in the same directory which makes it easy to support and understand.

A single level directory has a significant limitation, however, when the number of files increases or when the system has more than one user. All the files are in the same directory, they must have the unique name .If two users call their dataset test, then the unique name rule violated.

Two-level directory:

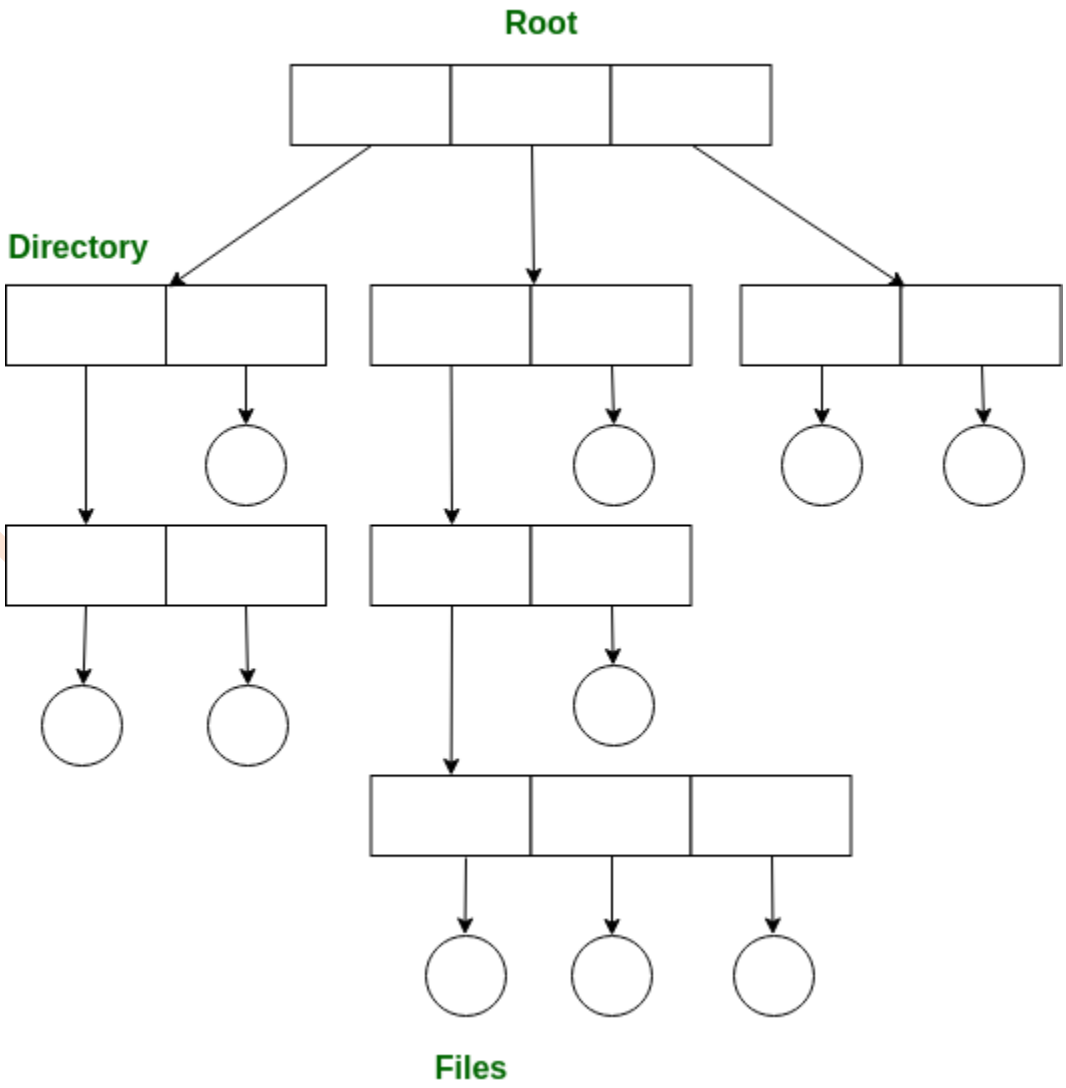
A single level directory often leads to confusion of file names among different users. The solution to this problem is to create a separate directory for each user.

In the two-level directory structure, each user has there own *user files directory (UFD)*. The UFDs has similar structures, but each lists only the files of a single user. system's *master file directory (MFD)* is searches whenever a new user id=s logged in. The MFD is indexed by username or account number, and each entry points to the UFD for that user.



Tree-structured directory:

Once we have seen a two-level directory is seen as a tree of height 2, the natural generalization is to extend the directory structure to a tree of arbitrary height. This generalization allows the user to create their own subdirectories and to organize on their files accordingly.



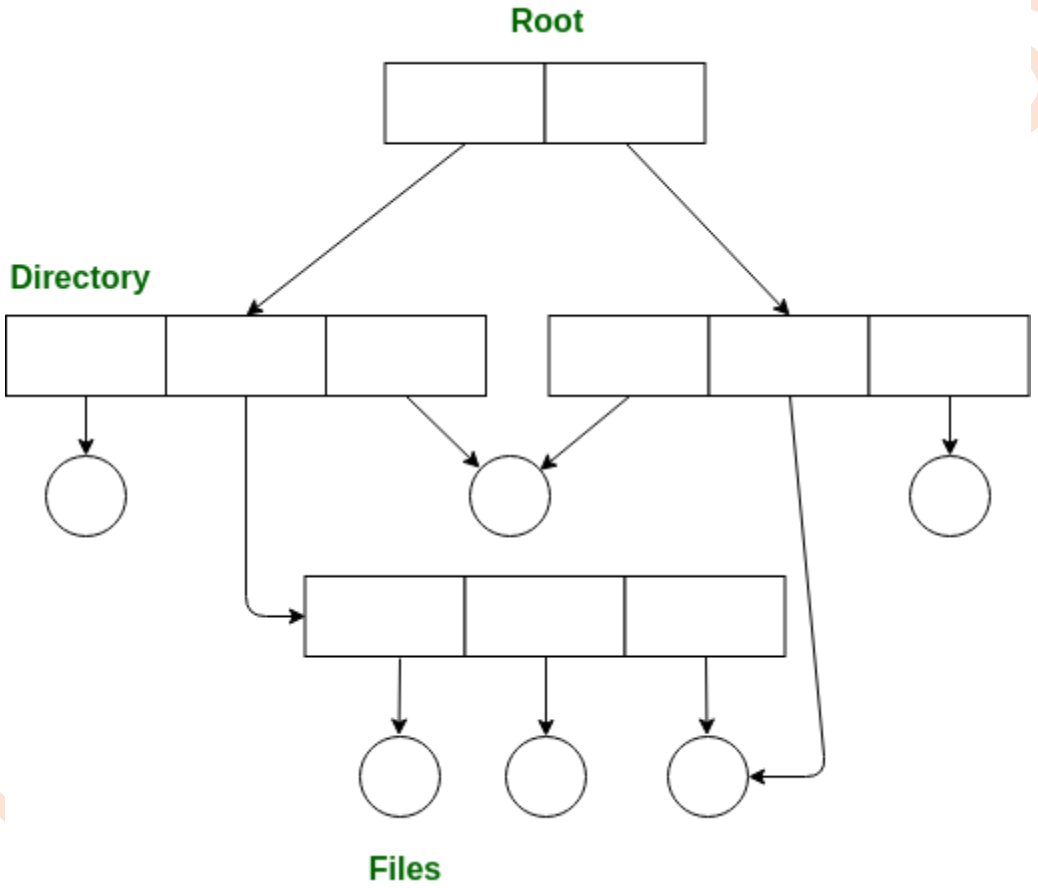
A tree structure is the most common directory structure. The tree has a root directory, and every file in the system have a unique path.

Acyclic graph directory:

An acyclic graph is a graph with no cycle and allows to share subdirectories and files. The same file or subdirectories may be in two different directories. It is a natural generalization of the tree-structured directory.

It is used in the situation like when two programmers are working on a joint project and they need to access files. The associated files are stored in a subdirectory, separating them from other projects and files of other programmers, since they are working on a joint project so they want the subdirectories to be into their own directories. The common subdirectories should be shared. So are used Acyclic directories.

It is to be noted that shared file is not the same as copy file. If any programmer makes some changes in the subdirectory it will reflect in both subdirectories.



General graph directory structure:

In general graph directory structure, cycles are allowed within a directory structure where multiple directories can be derived from more than one parent directory.

The main problem with this kind of directory structure is to calculate total size or space that has been taken by the files and directories.

Disk Scheduling Algorithms

Disk scheduling is done by operating systems to schedule I/O requests arriving for the disk. Disk scheduling is also known as I/O scheduling.

Importance disk scheduling:

- Multiple I/O requests may arrive by different processes and only one I/O request can be served at a time by the disk controller. Thus other I/O requests need to wait in the waiting queue and need to be scheduled.
- Two or more request may be far from each other so they can result in greater disk arm movement.
- Hard drives are one of the slowest parts of the computer system and thus need to be accessed in an efficient manner.

There are many Disk Scheduling Algorithms but before discussing them let's have a quick look at some of the important terms:

Seek Time:

Seek time is the time taken to locate the disk arm to a specified track where the data is to be read or writtern. So the disk scheduling algorithm that gives minimum average seek time is better.

Rotational Latency:

Rotational Latency is the time taken by the desired sector of disk to rotate into a position so that it can access the read/write heads. So the disk scheduling algorithm that gives minimum rotational latency is better.

Transfer Time:

Transfer time is the time to transfer the data. It depends on the rotating speed of the disk and number of bytes to be transferred.

Disk Access Time:

Disk Access Time is:

```
Disk Access Time = Seek Time +
                   Rotational Latency +
                   Transfer Time
```

Disk Response Time:

Response Time is the average of time spent by a request waiting to perform its I/O operation. *Average Response time* is the response time of all the requests. *Variance Response Time* is the measure of how individual request are serviced with respect to average response time. So the disk scheduling algorithm that gives minimum variance response time is better.

Disk Scheduling Algorithms:

FCFS:

FCFS is the simplest of all the Disk Scheduling Algorithms. In FCFS, the requests are addressed in the order they arrive in the disk queue.

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.

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.

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).

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 is occurred due to unnecessary traversal to the end of the disk.

CLOOK:

As LOOK is similar to SCAN algorithm, in similar way, CLOOK is similar to CSCAN disk scheduling algorithm. In CLOOK, the disk arm in spite of going to the end goes only to the last request to be serviced in front of the head and then from there goes to the other end's last request. Thus, it also prevents the extra delay which is occurred due to unnecessary traversal to the end of the disk.

RAID (Redundant Arrays of Independent Disks)

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 takes 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, the data can be retrieved and can 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.

Different RAID levels

RAID-0 (Striping)

Blocks are "striped" 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

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.

RAID-4 (Block-Level Striping 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.

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 Striping 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.

What about the other RAID levels?

RAID-2 consists of bit-level striping using a Hamming Code parity. RAID-3 consists of byte-level striping with a dedicated parity. These two are less commonly used. RAID-6 is a recent advancement which contains a distributed double parity, which involves block-level striping with 2 parity bits instead of just 1 distributed across all the disks. There are also hybrid RAIDs, which make use of more than one RAID levels nested one after the other, to fulfill specific requirements.