# OPERATING SYSTEMS

These notes have been prepared mainly from the following books :-

(1) Operating System Concepts
by Silberschatz, Galvin, Gagne

Kindly refer above book for more details.

# Processes

## Process :-

A process is a program in execution. A program by itself is not a process; a program is a passive entity, whereas a process is an active entity.

A process includes the following :-

(i) Text Section :-
It contains the program code.

(ii) Current Activity :-
It is represented by the value of the program counter and the contents of the processor's registers.

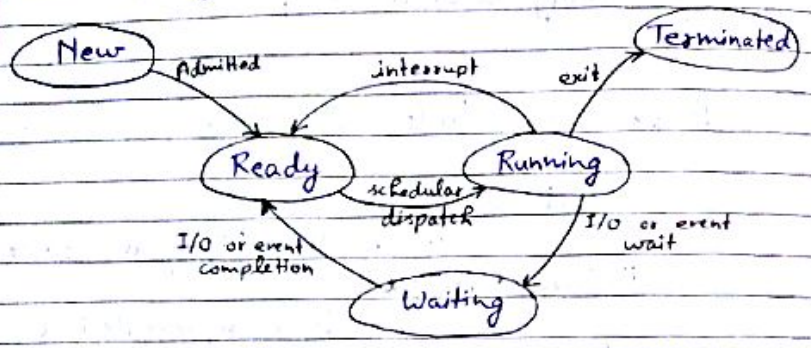
(iii) Stack :-
Contains temporary data such as function parameters, return addresses, and local variables.

(iv) Data Section :-
Contains global variables.

(v) Heap :-
Used for dynamic memory allocation during process runtime.

## Process State :-

Each process may be in one of the following states during execution :-




(i) New :- The process is being created

(ii) Running :- Instructions are being executed.

(iii) Waiting :- The process is waiting for some event to occur such as an I/O completion or reception of a signal.

(iv) Ready :- The process is waiting to be assigned to a processor.

(v) Terminated :-
The process has finished execution.

# Only one process can be running on any processor at any instant.

## Process Control Block :-

Each process is represented by a process control block (PCB) - also called a task control block.

A PCB contains the following information :-

(1) Process State
(2) Program Counter
(3) CPU Registers
(4) CPU Scheduling Information
(5) Memory Management Information
(6) Accounting Information
(7) I/O Status.
(8) Process Identifier (PID) & Parent Process Id (PPID)
(9) Address space for the process.
(10) User Identification
(11) List of open files.

## Operations on Processes :-

(i) Process Creation :-

A process may create several new processes, via a create-process system call, during the course of execution.

The creating process is called a parent process, and the new processes are called the children of that process. Each of these processes may in turn create other processes, forming a tree of processes.

When a process creates a subprocess, the sub-process may be able to obtain its resources directly from the operating system, or it may be constrained to a subset of the resources of the parent process.

Restricting a child process to a subset of the parent's resources prevents any process from overloading the system by creating too many subprocesses.

# In UNIX, a new process is created by the fork() system call.

# In UNIX, the exec() system call is used after a fork() system call to replace the process's memory space with a new program.

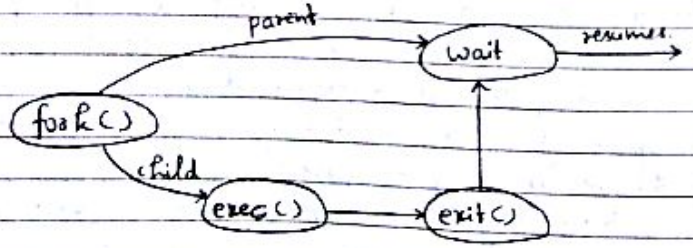
# In Win32 Api, new process is created using CreateProcess() function.

(ii) Process Termination :-

A process terminates when it finishes executing its final statement and asks the operating system to delete it by using the exit() system call.

Some systems do not allow a child to exist if its parent has terminated. In such systems, if a process terminates, then all its children must also be terminated. This phenomenon is referred to as Cascading termination.

# In UNIX, if the parent terminates, all its children are assigned init process as their parent.




Process Table :-

The process table is a data structure maintained by the operating system to facilitate context switching and scheduling.

Every process has an entry in process table. These entries are known as process control blocks (PCB).

Trap or Exception :-

A trap is a software generated interrupt caused either by an error or by a specific request from a user program.

Dual-Mode Operation in an Operating System :-

There are two seperate modes of operation in an operating system :-

(i) User Mode
(ii) Kernel Mode or Supervisor Mode or System Mode or Privileged Mode.

Mode Bit :-

It is used to distinguish between user mode and kernel mode.

(a) For user mode :- mode-bit = 1
(b) For kernel mode :- mode-bit = 0

# At system boot time, the hardware starts in kernel mode. The operating system is then loaded and starts user applications in user mode.

# Whenever a trap or interrupt occurs, the hardware switches from user mode to kernel mode.

# The dual mode of operation provides us with the means for protecting the operating system from errant users and errant users from one another.

# Some of the machine instructions that may cause harm are designated as privileged instructions. The hardware allows privileged instructions to be executed only in kernel mode.

# If an attempt is made to execute a privileged instruction in user mode, the hardware treats it as illegal and traps it to the operating system.

# Switching to user mode, I/O Control, timer management and interrupt management are privileged instructions.

# When a system call is executed, it is treated by the hardware as software interrupt.

# System calls provide the means for a user program to ask the operating system to perform tasks reserved for the operating system on the user program's behalf.

# Dual Mode Operation support is provided by the hardware not the software.

Timer :-

→ Timer is used to ensure that the operating system maintains control over the CPU.

→ A timer can be set to interrupt the computer after a specified period.

→ We can use the timer to prevent a user program from running too long.

Fork() :-

When a process forks, it creates a copy of itself.

In the child process, the return value of the fork is zero.

In the parent process, the return value is the PID of the newly created child process.

The fork operation creates a separate address space for the child.

Both the parent and child processes posses the same code segments, but execute independently of each other.

```
for (int i=0; i<n; i++)
    fork()
```

For this program $(2^n - 1)$ childs will be created.

# Multithreading

## Problems with traditional processes :-

(i) Traditional processes have separate address spaces.
(ii) Process creation is expensive.
(iii) Process switch is expensive.
(iv) Sharing memory areas among processes is non-trivial.
(v) A process is a single thread of execution.

## Threads :-

A thread is a basic unit of CPU utilization. It comprises :-

(i) Thread ID
(ii) Program Counter (PC)
(iii) A register set
(iv) A stack

It shares with other threads belonging to the same process its code section, data section, and other operating-system resources, such as open files and signals.

# Many operating system kernels are now multithreaded

Benefits of Multithreading :-

(i) Responsiveness :-
Multithreading an interactive application may allow

Benefits of Multithreading :-

(i) Responsiveness :-
Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation thereby increasing responsiveness to the user.

(ii) Resource Sharing :-
Threads share the address space and the resources of the process to which they belong.

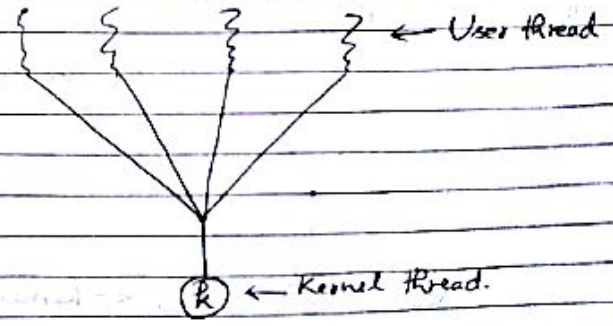
(iii) Economy :-
(a) Thread creation is much simpler than process creation.
(b) Thread switch is simple.
(c) Communication b/w threads is simple.

(iv) Utilization of multiprocessor architectures :-
Threads can run in parallel on different processors.

Types of Threads :-
Two types :-
(1) User-level threads :- (managed by thread library)
Kernel not aware of threads.
(2) Kernel-level threads :-
All thread-management done in kernel.

Multithreading Models :-
There must exist a relationship between user threads and kernel threads. There are three common ways of establishing this relationship :-

(i) Many-to-One Model :-
It maps many user-level threads to one kernel thread.
It is efficient, but the entire process will block if a thread makes a blocking system call.
Because only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multiprocessors.



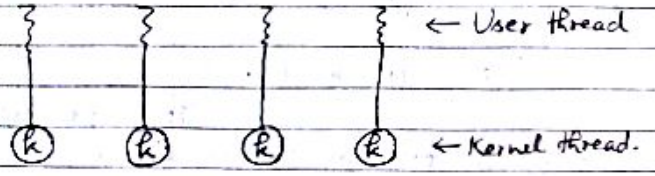
(ii) One-to-One Model :-

It maps each user thread to a kernel thread.

It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call.

It also allows multiple threads to run in parallel on multiprocessors.

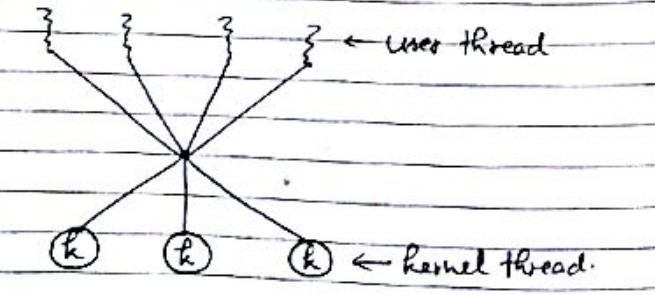
The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread.




(iii) Many-to-Many Model :-

It multiplexes many user-level threads to a smaller or equal number of kernel threads.

It does not suffer from the shortcomings of other two models.




Pop-Up Threads :-

In distributed systems, arrival of a message causes the system to create a new thread to handle the message. Such a thread is called a pop-up thread.

A key advantage of pop-up threads is that since they are brand new, they do not have any history - registers, stacks etc. that must be restored. Each one starts out fresh and each one is identical to all the others.

# Process Scheduling

## Multiprogramming :-

The objective of multiprogramming is to have some process running at all times, to maximise CPU utilization.

In multiprogrammed systems, the user can not interact with the system.

## Time Sharing :-

The objective of time sharing is to switch the CPU among processes so frequently that users can interact with each program while its running.

Time sharing system are also called as multitasking systems.

For a time sharing system, the response time should be short.

Time sharing systems internally use multiprogramming

## Scheduling Queues:-

(i) Job Queue:-

As processes enter the system, they are put into a job queue, which consists of all processes in the system.

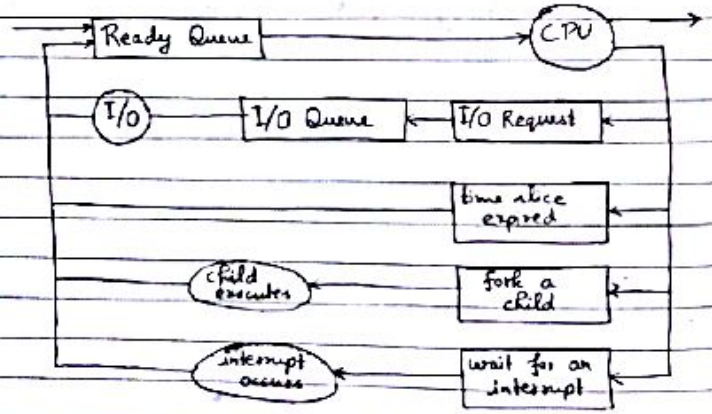
(ii) Ready Queue:-

The processes that are residing in main memory and are ready and waiting to execute are kept on a list called the ready queue.

(iii) Device Queue:-

The list of processes waiting for a particular I/O device is called a device queue.

Each device has its own device queue.




(Queueing-diagram representation of process scheduling)

## Schedulers:-

A process migrates among the various scheduling queues throughout its lifetime. The schedulers are used to select the processes from these queues in some fashion.

There are generally three types of schedulers:-

(1) Long-term Scheduler or Job Scheduler:-

Often, in a batch system, more processes are submitted than can be executed immediately. These processes are spooled to a mass-storage device, where they are kept for later execution.

The long-term scheduler selects processes from this pool and loads them into memory for execution.

The long-term scheduler executes much less frequently.

The long-term scheduler controls the degree of multiprogramming (the number of processes in memory). If the degree of multiprogramming is stable, then the average rate of process creation must be equal to the average departure rate of processes leaving the system.

The long-term scheduler selects a good process mix of I/O-bound and CPU-bound processes.

The long-term scheduler may need to be invoked only when a process leaves the system. Because of the longer interval between executions, the long-term scheduler can afford to take more time to decide which process should be selected for execution.

(2) Short-term scheduler or CPU Scheduler :-

The short-term scheduler selects from among the processes that are ready to execute and allocates the CPU to one of them.

The short-term scheduler must select a new process for the CPU frequently and it must be fast.

(3) Medium-term Schedules :-

The key idea behind a medium term schedules is that sometimes it can be advantageous to remove processes from memory and thus reduce the degree of multiprogramming. Later, the process can be reintroduced into memory, and its execution can be continued where it left-off.

This scheme is called swapping. The process is swapped out, and is later swaped-in, by the medium term schedules.

# On some systems, the long-term schedules may be absent or minimal, like some time-sharing systems.

# Medium-term Scheduler is generally used in time-sharing systems.

Context Switch :-

Switching the CPU to another process requires performing a state save of the current process and a state restore of a different process. This task is known as a context switch.

Context-switch time is pure overhead, because system does no useful work while switching.

~~Its speed depends on~~

Its speed varies from machine to machine, depending on the memory speed, the number of registers that must be copied, and the existence of special instruction (such as a single instruction to load or store all registers).

Dispatcher :-

The dispatcher is the module that gives control of the CPU to the process selected by the short-term scheduler.

This function involves the following :-

(i) Switching context

(ii) Switching to user mode

(iii) Jumping to the proper location in the user program to restart the program.

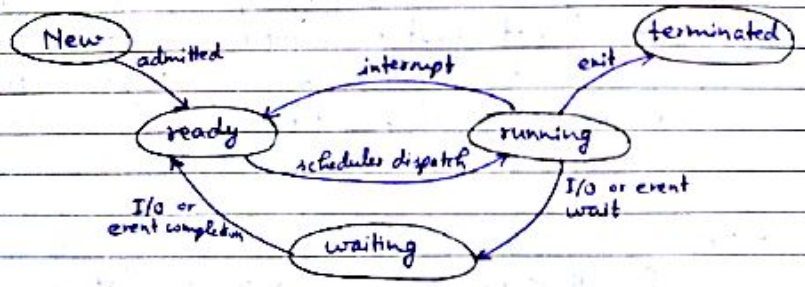
The dispatcher should be as fast as possible.

The time taken by the dispatcher to stop one process and start another running is known as the dispatch latency.

## Preemptive and Nonpreemptive Scheduling :-

→ Under nonpreemptive scheduling, once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to a the waiting state.

→ Nonpreemptive scheduling is also called as Cooperative scheduling.




→ Whenever a process switches from the running state to the ready state or waiting state to the ready state, and if scheduling takes place in these cases then the scheduling is called preemptive scheduling.

→ A scheduling is preemptive if once a process has been given the CPU can taken away.

→ Preemptive scheduling incurs a cost associated with access to shared data. The shared data can become inconsistent, if one process that is updating some data is preempted by another process and after that another process tries to read this data.

## Scheduling Criteria :-

(i) CPU Utilization :-
CPU should be as busy as possible.

(ii) Throughput :-
The number of processes that are completed per unit time, is called throughput.

(iii) Turnaround time :-
The interval from the time of submission of a process to the time of completion is the turnaround time.

(iv) Waiting time :-
Waiting time is the sum of the periods spent waiting in the ready queue

(v) Response time :-
The time from the submission of a request until the first response is produced is called response time.

# It is desirable to maximize CPU utilization and throughput and to minimize turnaround time, waiting time, and response time.

## Scheduling Algorithms :-

### (1) First-Come, First Served Scheduling :-

The process that requests the CPU first is allocated to the CPU first. Its implemented using FIFO Queue.

FCFS scheduling algorithm is non-preemptive. The average waiting time in FCFS is often quite long.

### (2) Shortest-Job-First Scheduling :-

This algorithm associates with each process the length of the process's next CPU burst. When the CPU is available, it is assigned to the process that has the smallest next CPU burst.

If the next CPU bursts of two processes are same, the FCFS scheduling is used to break the tie.

The SJF algorithm is optimal, in that it gives the minimum average waiting time for a given set of processes.

The SJF algorithm can not be implemented at the level of short-term CPU scheduling. There is no way to know the length of the next CPU burst.

The SJF algorithm can be either preemptive or nonpreemptive.

Preemptive SJF scheduling is sometimes called Shortest-Remaining-Time-First Scheduling.

### (3) Priority Scheduling :-

A priority is associated with each process, and the CPU is allocated to the process with the highest priority.

Equal-priority processes are scheduled in FCFS order.

Priority scheduling can be either preemptive or nonpreemptive.

A preemptive priority scheduling algorithm will preempt the CPU if the priority of the newly arrived process is higher than the priority of the currently running process.

A major problem with priority scheduling algorithms is indefinite blocking or starvation.

A solution to the problem of indefinite blockage of low-priority processes is aging.

### Aging :-

Aging is a technique of gradually increasing the priority of processes that wait in the system for a long time.

## (4) Round-Robin Scheduling :-

The round-robin scheduling algorithm is designed especially for time-sharing systems.

It is similiar to FCFS scheduling, but preemption is added to switch between processes.

The ready queue is treated as a circular queue. The CPU scheduler goes around the ready queue, allocating the CPU to each process for a time interval of up to 1 time quantum.

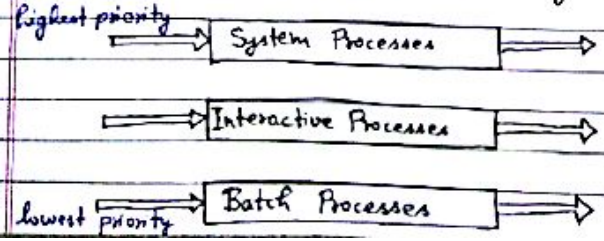
## (5) Multilevel Queue Scheduling :-

A multilevel queue scheduling algorithm partitions the ready queue into several separate queues.

The processes are permanently assigned to one queue, generally based on the properties of the process, such as memory size, process priority, or process type.

Each queue has its own scheduling algorithm. For example, separate queues might be used for foreground and background processes, and these processes might have separate scheduling requirements.

In addition, there must be scheduling among the queues, which is commonly implemented as fixed-priority preemptive scheduling.

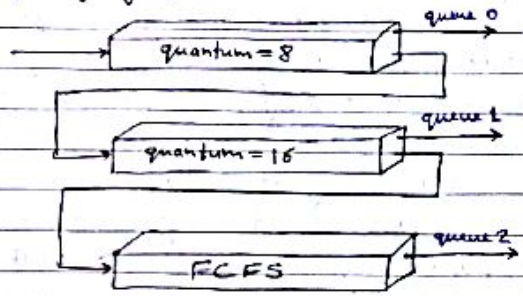



## (6) Multilevel Feedback-Queue Scheduling :-

The multilevel feedback-queue scheduling algorithm, in contrast, allows a process to move between queues.

The idea is to separate processes according to the characteristics of their CPU bursts. If a process uses too much CPU time, it will be moved to a lower-priority queue.

This scheme leaves I/O-bound and interactive processes in the higher priority queues. In addition, a process that waits too long in a lower-priority queue may be moved to a higher-priority. This form of aging prevents starvation.




A process entering the ready queue is put in queue 0. A process in queue 0 is given a time quantum of 8 ms. If it does not finish within this time, it is moved to the tail of queue 1. If queue 0 is empty, the process at the head of queue 1 is given a quantum of 16 ms. If it does not complete, it is preempted and is put in queue 2. Process in queue 2 are run in on FCFS basis but are run only when queue 0 and 1 are empty.

## Multiple-Processor Scheduling :-

There are two approaches to Multiple-Processor Scheduling :-

(1) Asymmetric Multiprocessing :-

All scheduling decisions, I/O processing, and other system activities handled by a single processor - the master server.

(2) Symmetric Multiprocessing (SMP) :-

Here each processor is self scheduling.

## Processor Affinity :-

When a process migrates from one processor to another processor, the contents of the cache memory must be invalidated for the processor being migrated from, and the cache must be repopulated for the processor being migrated to.

Because of the high cost of invalidating and re-populating caches, most SMP systems try to avoid migration of processes and instead attempts to keep a process running on the same processor.

This is known as processor affinity.

## Load Balancing :-

Load balancing attempts to keep the workload evenly distributed across all processors in an SMP system. The load balancing is typically only necessary on systems where each processor has its own private queue of eligible processes to execute.

There are two approaches to load balancing :-

(1) Push Migration :-

A specific task periodically checks the load on each processor and - if it finds an imbalance - evenly distributes the load by moving processes from overloaded to idle or less busy processors.

(2) Pull Migration :-

It occurs when an idle processor pulls a waiting task from a busy processor.

# Push and Pull migration need not be mutually exclusive and are often implemented in parallel.

# Deadlocks

## Deadlock :-

A set of processes is in a deadlock state when every process in the set is waiting for an event that can be caused only by another process in the set.

## Necessary Conditions for deadlock :-

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

(1) Mutual Exclusion :-

At least one resource must be held in a nonsharable mode; that is, only one process at a time can use the resource.

(2) Hold and Wait :-

A process must be holding at least one resource and waiting for acquiring additional resources that are currently being held by other processes.

(3) No preemption :-

Resources can not be preempted; that is, a resource can be released only voluntarily by the process holding it, after that process has completed its task.

(4) Circular Wait :-

A set $\{P_0, P_1, \ldots P_n\}$ of waiting process must exist such that $P_0$ is waiting for a resource held by $P_1$; $P_1$ is waiting for a resource held by $P_2$, $\ldots$, $P_{n-1}$ is waiting for a resource held by $P_n$ and $P_n$ is waiting for a resource held by $P_0$.

These four conditions are not completely independent.

## Resource-Allocation Graph :-

→ Used for describing deadlocks precisely.

→ This graph consists of a set of vertices V and a set of edges E. The set of vertices V is partitioned into two different types of nodes :-

(i) $P = \{P_1, P_2, \ldots P_n\}$ :- The set consisting of all the active processes in the system.

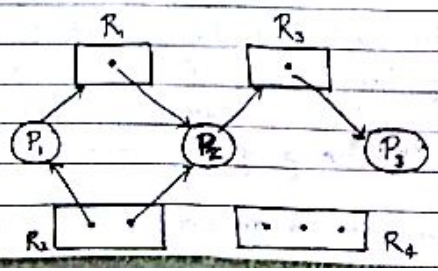
(ii) $R = \{R_1, R_2, \ldots R_m\}$ :- The set consisting of all resource types in the system.

**Request Edge :-**

A directed edge from process $P_i$ to resource $R_j$, denoted by $P_i \rightarrow R_j$, is called a request edge. It signifies that process $P_i$ has requested an instance of resource type $R_j$ and is currently waiting for that resource.

**Assignment Edge :-**

A directed edge from resource type $R_j$ to process $P_i$, denoted by $R_j \rightarrow P_i$, is called an assignment edge. It signifies that an instance of resource type $R_j$ has been allocated to process $P_i$.



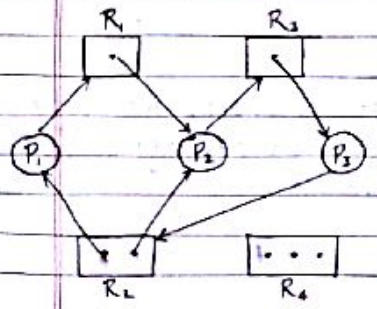

Resource-allocation Graph

# If the graph contains no cycles, then no process in the system is deadlocked.

# If the graph does contain a cycle, then a deadlock may exist.

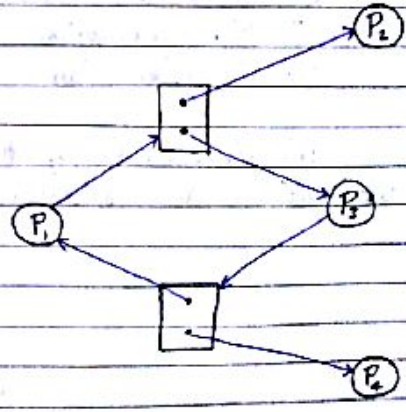
# If the cycle involves only a set of resource types, each of which has only a single instance, then a deadlock has occurred.

# If each resource type has several instances, then a cycle does not necessarily imply that a deadlock has occured.




Resource-allocation graph with a deadlock

Resource-allocation graph with a cycle but no deadlock



## Methods for Handling Deadlocks :-

There are three ways to deal with the deadlock problem :-

(i) We can use a protocol to prevent or avoid deadlocks, ensuring that the system will never enter a deadlock state.

(ii) We can allow the system to enter a deadlock state, detect it, and recover.

(iii) We can ignore the problem altogether and pretend that deadlocks never occur in the system.

## Deadlock Prevention :-

By ensuring that at least one of these conditions can not hold, we can prevent the occurance of a deadlock :-

(i) Mutual Exclusion
(ii) Hold and Wait
(iii) No preemption
(iv) Circular Wait

The drawback of this method is low device utilization and reduced system throughput.

## Deadlock Avoidance :-

A deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that a circular wait condition can never exist.

The resource-allocation state is defined by the number of available and allocated resources and the maximum demand of the processes.

There are two algorithms for deadlock-avoidance :-
(i) Resource-Allocation-Graph Algorithm.
(ii) Banker's Algorithm.

## Safe State :-

A state is safe if the system can allocate resources to each process in some order and still avoid a deadlock. More formally, a system is in a safe state only if there exists a safe sequence.

A sequence of processes $\langle P_1, P_2, \ldots, P_n \rangle$ is a safe sequence for the current allocation state if, for each $P_i$, the resource requests that $P_i$ can still make can be satisfied by the currently available resources plus the resources held by all $P_j$, with $j < i$.

# A safe state is not a deadlocked state. Conversely, a deadlocked state is an unsafe state.

# Not all unsafe states are deadlocks, however An unsafe state may lead to a deadlock.

Initially, the system is in a safe state. Whenever a process requests a resource that is currently available the system must decide whether the resource can be allocated immediately or whether the process must wait. The request is granted only if the allocation leaves the system in a safe state.

Resource-Allocation-Graph Algorithm :-

This algorithm is used when each resource has only one instance.

Here we use a new type of edge, called a claim edge.

A claim edge $P_i \rightarrow R_j$ indicates that process $P_i$ may request resource $R_j$ at some time in the future. This edge resembles a request edge in direction but is represented in the graph by a dashed line.

When process $P_i$ requests resource $R_j$, the claim edge $P_i \rightarrow R_j$ is converted to a request edge.

Similiarly when a resource $R_j$ is released by $P_i$, the assignment edge $R_j \rightarrow P_i$ is reconverted to a claim edge $P_i \rightarrow R_j$.

Suppose that process $P_i$ requests resource $R_j$. The request can be granted only if converting the request edge $P_i \rightarrow R_j$ to an assignment edge $R_j \rightarrow P_i$ does not result in the formation of a cycle in the resource allocation graph.

Banker's Algorithm :-

This algorithm is used when a resource type may have multiple instances.

When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need. This number may not exceed the total number resources in the system.

When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state. If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources.

Let n be the number of processes in the system and m be the number of resource types. We need the following data structures :-

(i) Available :-

A vector of length m indicates the number of available resources of each type.

If Available[j] equals k, there are k-instances of resource type $R_j$ available.

(ii) Max :-

An n x m matrix defines the maximum demand of each process.

If Max[i][j] = k, then process $P_i$ may request at most k-instances of resource type $R_j$.

(iii) Allocation :-

An n x m matrix defines the number of resources of each type currently allocated to each process.

If Allocation[i][j] = k, then process $P_i$ is currently allocated k instances of resource type $R_j$.

(iv) Need :-

An n x m matrix indicates the remaining resource need of each process.

If Need[i][j] = k, then process $P_i$ may need k more instances of resource type $R_j$ to complete its task.

Need[i][j] = Max[i][j] - Allocation[i][j]

Example :- Consider a system with five processes $P_0$ through $P_4$ and three resource types A, B and C. Resource type A has 10 instances, B has 5 instances, and C has 7 instances. Suppose that at time $T_0$, the following snapshot of the system has been taken :-

| | Allocation | Max | Available |
|---|---|---|---|
| | A B C | A B C | A B C |
| $P_0$ | 0 1 0 | 7 5 3 | 3 3 2 |
| $P_1$ | 2 0 0 | 3 2 2 | |
| $P_2$ | 3 0 2 | 9 0 2 | |
| $P_3$ | 2 1 1 | 2 2 2 | |
| $P_4$ | 0 0 2 | 4 3 3 | |

Now check :- (i) Whether the system is in safe state or not?
(ii) Whether Request (1,0,2) by $P_1$ can be immediately granted?
(iii) Whether Request (3,3,0) by $P_4$ can be immediately granted?

Solution :- The contents of the matrix Need are :-

| | Need | |
|---|---|---|
| | A B C | (Need = Max - Allocation) |
| $P_0$ | 7 4 3 | |
| $P_1$ | 1 2 2 | |
| $P_2$ | 6 0 0 | |
| $P_3$ | 0 1 1 | |
| $P_4$ | 4 3 1 | |

(i) Since the sequence $<P_1, P_3, P_4, P_2, P_0>$ satisfies the safety criteria. So, the system is currently in a safe state.

(ii) Request = (1, 0, 2) by $P_1$.

To decide whether this request can be immediately granted, we first check that Request ≤ Available - that is, that (1, 0, 2) ≤ (3, 3, 2) which is true.

We then pretend that this request has been fulfilled, and we arrive at the following new state :-

| | Allocation | Need | Available |
|---|---|---|---|
| $P_0$ | 1 1 2 | 6 4 1 | 2 3 0 |
| $P_1$ | 2 0 0 | 1 2 2 | |
| $P_2$ | 3 0 2 | 6 0 0 | |
| $P_3$ | 2 1 1 | 0 1 1 | |
| $P_4$ | 0 0 2 | 4 3 1 | |
| | A B C | A B C | A B C |

We must determine whether this new system is safe or not. Here sequence $<P_1, P_3, P_4, P_0, P_2>$ satisfies the safety requirement. Hence, we can immediately grant the request of process $P_1$.

(iii) Request (3, 3, 0) by $P_0$.

Here Request ≤ Available
⇒ (3, 3, 0) ≤ (3, 3, 2)

Now suppose this request has been fulfilled and we arrive at the following new state :-

| | Allocation | | | Need | | | Available | | |
|---|---|---|---|---|---|---|---|---|---|
| | A | B | C | A | B | C | A | B | C |
| $P_0$ | 0 | 1 | 0 | 7 | 4 | 3 | 0 | 0 | 2 |
| $P_1$ | 2 | 0 | 0 | 3 | 2 | 2 | | | |
| $P_2$ | 3 | 0 | 2 | 9 | 0 | 2 | | | |
| $P_3$ | 2 | 1 | 1 | 2 | 2 | 2 | | | |
| $P_4$ | 3 | 3 | 4 | 1 | 0 | 1 | | | |

Since this new system is not safe. Hence we can not grant this request of process $P_4$.



## Recovery from Deadlock :-

When a deadlock detection algorithm determines that a deadlock exists, several alternatives are available :-

(1) Process termination :-

Two methods are available :-

(a) Abort all deadlocked processes.

(b) Abort one process at a time until the deadlock cycle is eliminated.

(2) Resource Preemption :-

To eliminate deadlocks using resource preemption, we successfully preempt some resources from processes and give these resources to other processes until the deadlock cycle is broken.

In this method, three issues need to be addressed :-

(1) Selecting a victim :-

Which resources and which processes are to be preempted ?

(2) Rollback :-

If we preempt a resource from a process, we must rollback the process to some safe state and restart it from that state.

(3) Starvation :-

How can we guarantee that resources will not always be preempted from the same process.

# Interprocess Communication & Synchronization

## Interprocess Communication :-

Processes executing concurrently in the operating system may be either independent processes or cooperating process.

A process is independent if it can not affect or be affected by the other processes executing in the system. Any process that does not share data with any other process is independent.

A process is cooperating if it can affect or be affected by the other processes executing in the system. Clearly, any process that shares data with other processes is a cooperating process.

There are several reasons for providing an environment that allows process cooperation :-

(i) Information Sharing

(ii) Computation Speedup :-

If we want a particular task to run faster, we must break it into subtasks, each of which will be executing in parallel with the others.

(iii) Modularity :-

We may want to construct the system in a modular fashion, dividing the system functions into seperate processes or threads.

(iv) Convenience :-

Even an individual user may work on many tasks at the same time.

Cooperating processes require an interprocess communication (IPC) mechanism that will allow them to exchange data and information.

There are two fundamental models of inter-process communication :-

(1) Shared Memory.
(2) Message Passing.

Shared Memory Model :-

In this model, a region of memory that is shared by cooperating processes is established.

Typically, a shared-memory region resides in the address space of the process creating the shared memory segment. Other processes that wish to communicate using this shared-memory segment must attach it to their address space.

The processes are also responsible for ensuring that they are not writing to the same location simultaneously.

Message-Passing Systems :-

Message passing provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space and is particularly useful in a distributed environment where the communicating processes may reside on different computers connected by a network.

A message-passing facility provides at least two operations :-

(i) Send (message)
(ii) Receive (message)

The communication between different processes in a message passing system can be of following types :-

(a) Synchronous or Asynchronous Communication
(b) Direct or Indirect Communication.

Synchronous Communication :-

In synchronous communication processes synchronize at every message. Both send and receive are blocking operations.

Asynchronous Communication :-

The send operation is almost always non-blocking. The receive operation, however, can be blocking or non-blocking.

Blocking Send :-
The sending process is blocked until the message is received by the receiving process.
Nonblocking Send :-
The sending process sends the message and resumes operation.
Blocking Receive :-
The receiver blocks until a message is available.
Nonblocking Receive :-
The receiver retrieves either a valid message or a null.

Direct Communication :-
Processes must explicitly name the receiver or sender of a message (symmetric addressing).
- send (P, message). Send message to process P.
- receive (Q, message). Receive message from Q.

In a client-server system, the server does not have to know the name of a specific client in order to receive a message. In this case, a variant of the receive operation can be used (asymmetric addressing).

- receive (ID, message). Receive a pending message from any process. When a message arrives, ID is set to the name of the sender.

In direct communication, the interconnection between the sender and receiver has the following characteristics :-
(i) A link is established automatically, but the processes need to know each other's identity.
(ii) A unique link is associated with the two processes.
(iii) The link is usually bi-directional, but it can be unidirectional.

Indirect Communication :-
With indirect communication, the messages are sent to and received from mailboxes, or ports.
A mailbox can be viewed abstractly as an object into which messages can be placed by processes and from which messages can be removed.
The send() and receive() premitives are defined as follows :-
- send (A, message). Send a message to mailbox A.
- receive (A, message). Receive a message from mailbox A.

Generally, a mailbox is associated with many senders and receivers.

In some systems, only one receiver is associated with a particular mailbox, such a mailbox is often called a port.

In indirect communication, the interconnection between the sender and receiver has the following characteristics:-

(i) A link is established between two processes only if they 'share' a mailbox.

(ii) A link may be associated with more than two processes.

(iii) Communicating processes may have different links between them, each corresponding to one mailbox.

Producer-Consumer Problem:-

A producer process produces information that is consumed by a consumer process.

One solution to the producer-consumer problem uses shared memory. To allow producer and consumer processes to run concurrently, we must have available a buffer of items that can be filled by the producer and emptied by the consumer.

The producer and consumer must be synchronized, so that the consumer does not try to consume an item that has not yet been produced.

Two types of buffers can be used:-

(i) Unbounded Buffer:-

It places no practical limit on the size of the buffer. The producer can always produce new items, but the consumer may have to wait for new items.

(ii) Bounded Buffer:-

It assumes a fixed buffer size. In this case, the consumer must wait if the buffer is empty, and the producer must wait if the buffer is full.

~~Solution of bounded buffer problem~~

## Solution of producer-consumer problem using bounded buffer :-

The following variables reside in a region of memory shared by the producer and consumer processes:-

```
① #define BUFFER_SIZE 10
② typedef struct {
        ...
    } item;
③ item buffer[BUFFER_SIZE];
④ int in = 0;
⑤ int out = 0;
```

The shared buffer is implemented as a circular array with two logical pointers :- in and out.

The variable in points to the next free position in the buffer; out points to the first full position in the buffer.

The buffer is empty when in == out; the buffer is full when ((in + 1) % BUFFER_SIZE) == out.

### Code for Producer :-

```
item nextProduced;
while (true) {
    /* produce an item in nextProduced */
    while (((in+1) % BUFFER_SIZE) == out)
    ;   /* do nothing */
    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
}
```

### Code for Consumer :-

```
item nextConsumed;
while (true) {
    while (in == out)
        ;   /* do nothing */
    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    /* consume the item in nextConsumed */
}
```

The producer process has a local variable nextProduced in which the new item to be produced is stored.

The consumer process has a local variable nextConsumed in which the item to be consumed is stored.

This scheme allows at most BUFFER_SIZE - 1 items in the buffer at the same time. Suppose we want to modify the algorithm to remedy this deficiency.

One possibility is to add an integer variable counter, initialized to 0. Counter is incremented every time we add a new item to the buffer and is decremented every time we remove one item from the buffer.

Modified code for producer :-

```
while (true)
{
  /* produce an item in nextPublished */
  while (counter == BUFFER_SIZE)
    ; /* do nothing */
  buffer[in] = nextProduced;
  in = (in + 1) % BUFFER_SIZE;
  counter ++;
}
```

Modified Code for Consumer :-

```
while (true)
{
  while (counter == 0)
    ; /* do nothing */
  nextConsumed = buffer[out];
  out = (out + 1) % BUFFER_SIZE;
  counter --;
  /* Consume the item in nextConsumed */
}
```

Although both the producer and consumer routines are correct separately, they may not function correctly when executed concurrently.

Suppose that the value of the variable counter is currently 5 and that the producer and consumer processes execute the statements "counter++" & "counter--" concurrently. Following the execution of these two statements, the value of the variable counter may be 4, 5, or 6!

~~The only correct result, though is counter == 5~~

The only correct result, though, is counter == 5, which is generated correctly if the producer and consumer execute separately.

Race Condition :-

A situation, where several processes access and manipulate the same data concurrently and the outcome of the execution depends on the particular order in which the access takes place, is called a race condition.

# To guard against the race condition above, we need to ensure that only one process at a time can be manipulating the variable counter. To make such a guarantee, we require that the processes be synchronized in some way.

## The Critical-Section Problem:-

Consider a system consisting of n processes $\{P_0, P_1, \ldots P_{n-1}\}$. Each process has a segment of code, called a critical section, in which the process may be changing common variables, updating a table, writing a file, and so on.

The important feature of the system is that, when one process is executing in its critical section, no other process is to be allowed to execute in its critical section.

The critical-section problem is to design a protocol that the processes can use to cooperate.

Each process must request permission to enter its critical section.

```
do {
    entry section
        critical section
    exit section
        remainder section
} while (true);
```

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

(1) Mutual Exclusion:-

If process $P_i$ is executing in its critical section, then no other processes can be executing in their critical sections.

(2) Progress:-

If no process is executing in its critical section and some processes wish to enter their critical sections, then only those processes that are not executing in their remainder sections can participate in the decision on which will enter its critical section next.

(3) Bounded Waiting:-

There exists a bound, or limit, on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

Two general approaches are used to handle critical sections in operating systems:-

(i) Preemptive Kernels:- Allows a process to be preempted while its running in kernel mode.

(ii) Nonpreemptive Kernels:- Does not allow a process running in kernel mode to be preempted. It will run until it exits kernel mode, blocks, or voluntarily yields control of the CPU.

A nonpreemptive kernel is essentially free from race conditions on kernel data structures, as only one process is active in the kernel at a time.

## Peterson's Solution :-

Peterson's solution is restricted to two processes that alternate execution between their critical sections and remainder sections. The processes are numbered $P_i$ and $P_j$, where $j = 1 - i$

Peterson's solution requires two data items to be shared between the two processes :-

```
int turn;
boolean flag[2];
```

The variable turn indicates whose turn it is to enter its critical section.

The flag array is used to indicate that a process is ready to enter its critical section.

### The structure of process $P_i$ in Peterson's solution :-

```
do {
    flag[i] = true;
    turn = j;
    while (flag[j] && turn == j);      (entry section)

    critical section

    flag[i] = false;                   (exit section)

    remainder section
} while (true);
```

To enter the critical section, process $P_i$ first sets flag[i] to be true and then sets turn to the value j, thereby asserting that if the other process wishes to enter the critical section, it can do so.

If both processes try to enter at the same time, turn will be set to both i and j at roughly the same time. Only one of these assignments will last, the other will occur but will be overwritten immediately. The eventual value of turn decides which of the two processes is allowed to enter its critical section first.

# Peterson's solution is a software based solution.

## Synchronization Hardware :-

In general, we can state that any solution to the critical-section problem requires a simple tool - a lock. Race conditions are prevented by requiring that critical regions be protected by locks.

```
do {
        acquire lock
            critical section
        release lock
            remainder section
} while (true);
```

The critical-section problem could be solved simply in a uniprocessor environment if we could prevent interrupts from occuring while a shared variable was being modified. In this manner, we could be sure that the current sequence of instructions would be allowed to execute in order without preemption. This is the approach taken by nonpreemptive kernels.

Unfortunately, this solution is not as feasible in a multiprocessor environment. Disabling interrupts on a multiprocessor can be time consuming.

Many modern computer systems therefore provide special hardware instructions that allow us either to test and modify the content of a word or to swap the contents of two words atomically.

### (i) TestAndSet() instruction :-

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

```
boolean TestAndSet (boolean *lock)
{
    boolean rv = *lock;
    *lock = true;
    return rv;
}
```

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

```
do {
    while (TestAndSet(&lock))
        ;   /* do nothing */

        Critical Section

    lock = false;

        Remainder Section

} while (true);
```

(ii) Swap() Instruction :-

The Swap() instruction operates on the contents of two words. Like the TestAndSet() instruction, it is executed atomically.

```
void Swap (boolean *lock, boolean *key)
{
    boolean temp = *lock;
    *lock = *key;
    *key = temp;
}
```

If the machine supports the Swap() instruction, then mutual exclusion can be provided as follows:-

```
do {
    key = true;
    while (key == true)
        Swap (&lock, &key);

    Critical Section

    lock = false;

    Remainder Section

} while (true);
```

A global boolean variable lock is declared and is initialized to false.

Each process has a local boolean variable key.

Bounded-waiting mutual exclusion with TestAndSet() instruction :-

Although previous algorithms satisfy the mutual-exclusion requirement, they do not satisfy the bounded-waiting requirement.

For the algorithm using the TestAndSet() instruction that satisfies all the critical-section requirements, the common data structures are :-

```
boolean waiting[n];
boolean lock;
```

These data structures are initialized to false.

```
do {
    waiting[i] = true;
    key = true;
    while (waiting[i] && key)
        key = TestAndSet (&lock);
    waiting[i] = false;

    Critical Section

    j = (i+1) % n;
    while ((j != i) && !waiting[j])
        j = (j+1) % n;

    if (j == i)
        lock = false;
    else
        waiting[j] = false;

    Remainder Section

} while (true);
```

## Semaphores :-

The various hardware-based solutions to the critical-section problem are complicated for the application programmers to use. To overcome this difficulty, we can use a synchronization tool called a semaphore.

A semaphore S is an integer variable that, apart from initialization, is accessed only through two standard atomic operations :- wait() and signal().

The wait() operation was originally termed P and signal() was originally called V.

The definition of wait() is as follows :-

```
wait(S){
    while S<=0
        ; /* No operation */
    S--
}
```

The definiton of signal() is as follows :-

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

## Binary Semaphores :-

The value of a binary semaphore can range only between 0 and 1. On some systems, binary semaphores are known as mutex locks, as they are locks that provide mutual exclusion.

We can use binary semaphore to deal with the critical-section problem for multiple processes. The n processes share a semaphore, mutex, initialized to 1.

```
do {
    waiting(mutex);

        Critical Section

    signal(mutex);

        Remainder Section

} while(true);
```

## Counting Semaphores :-

The value of a counting semaphore can range over an unrestricted domain.

Counting semaphores can be used to control access to a given resource consisting of a finite number of instances. The semaphore is initialized to the number of resources available. Each process that wishes to use a resource performs a wait() operation. When a process releases a resource, it performs a signal() operation.

When the count for the semaphore goes to 0, all resources are being used.

## Semaphore Implementation :-

The main disadvantage of the semaphore definition given here is that it requires busy waiting. Busy waiting wastes CPU cycles that some other process might be able to use productively. This type of semaphore is also called a spinlock because the process "spins" while waiting for the lock.

To overcome the need for busy waiting, we can modify the definition of the wait() and signal() semaphore operations.

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

An wait() operation, the process can block itself instead of engaging in busy waiting. The block operation places a process into a waiting queue associated with the semaphore, and the state of the process is switched to the waiting state.

```
wait (semaphore *S)
    {
     S -> value --;
     if (S -> value < 0)
        {
         add this process to S -> list;
         block ();
        }
    }
```

When some other process executes a signal() operation, the process is restarted by a wakeup() operation, which changes the process from the waiting state to the ready state. The process is then placed in the ready queue.

```
signal (semaphore *S)
    {
      S -> value++;
      if (S -> value <= 0)
        {
          remove a process P from S -> list;
          wakeup (P);
        }
    }
```

Note :- Although under the classical definition of semaphores with busy waiting the semaphore value is never negative, this implementation may have negative semaphore values. If the semaphore value is negative, its magnitude is the number of processes waiting on that semaphore.

The critical aspect of semaphores is that they be executed atomically. We must guarantee that no two processes can execute wait() and signal() operations on the same semaphore at the same time.

This is a critical-section problem, and in a single-processor environment, we can solve it by simply inhibiting interrupts during the time the wait() and signal() operations are executing.

In a multiprocessor environment, disabling interrupts on every processor can be a difficult task. Therefore SMP systems must provide alternative locking techniques – such as spinlocks – to ensure that wait() and signal() are performed atomically.

Note:-

It is important to admit that we have not completely eliminated busy waiting with this definition of the wait() and signal() operations. Here we have limited the busy waiting to the critical sections of the wait() and signal() operations, and these sections are short.

Deadlocks and Starvation:-

The implementation of a semaphore with a waiting queue may result in a situation where two or more processes are waiting indefinitely for an event that can be caused only by one of the waiting processes. The event in question is the execution of a signal() operation. When such a state is reached, these processes are said to be deadlocked.

| $P_0$ | $P_1$ |
|---|---|
| wait(S); | wait(Q); |
| wait(Q); | wait(S); |
| ⋮ | ⋮ |
| signal(S); | signal(Q); |
| signal(Q); | signal(S); |

Another problem related to deadlocks is indefinite blocking or starvation, which may occur if we add and remove processes in LIFO order.

www.ankurgupta.net

# File Systems

## File System Structure :-

A file system is used to allow the data to be stored, located, and retrieved easily.

The file system itself is generally composed of many different levels :-

Application Programs
⇓
Logical File System
⇓
File - organization module
⇓
Basic file system
⇓
I/O Control
⇓
Devices

## I/O Control :-

It consists of device drivers and interrupt handlers to transfer information between the main memory and the disk systems.

## Basic File System :-

It needs only to issue generic commands to the appropriate device driver to read and write physical blocks on the disk.

## File Organization Module :-

It knows about files and their logical blocks as well as physical blocks. It translates logical block addresses to physical block addresses.

Logical File System:-

It manages the metadata information. It manages the directory structure to provide the file-organization module with the information the latter needs, given a symbolic file name.

It maintains file structure via file-control-blocks

File Control Block (FCB):-

It contains information about the file, including ownership, permissions, and location of the file contents.

File systems used by various kind of OS:-

(i) UNIX → UNIX File System (UFS), which is based on Berkeley Fast File System (FFS).
(ii) Windows → FAT, FAT32, NTFS
(iii) Linux → Extended File System (ext2 & ext3)



Several on-disk and in-memory structures are used to implement a file system. These structures vary depending on the operating system and the file system but general principles apply.

On disk, the file system may contain following information:-

(i) Boot Control Block (per Volume):-

It contains information needed by the system to boot an OS from that volume.

In UFS, it is called the boot block.
In NTFS, it is called the partition boot sector.

(ii) Volume Control Block (per Volume):-

It contains volume details, such as the number of blocks in the partition, size of the blocks etc.

In UFS, it is called a superblock.
In NTFS, it is stored in master file table.

~~(iii) Directory Structure (per Volume):-~~

(iii) Directory Structure (per File system):-

It is used to organize the files.

In UFS, this includes file names and associated inode numbers.
In NTFS, it is stored in the master file table.

(iv) File Control Block - FCB (per file):-

It contain details about the file, including file permissions, ownership, size etc.

In UFS, this is called the inode block.
In NTFS, this information is actually stored within the master file table.

The in-memory information is used for both file-system management and performance improvement via caching. The data are loaded at mount time and discarded at dismount.

The structures may include the ones described below :-

(i) in-memory mount table :-
It contains information about each mounted volume.

(ii) in-memory directory-structure cache :-
It holds the directory information of recently accessed directories.

(iii) System-wide open-file table :-
It contains the FCB of each open file.

(iv) Per-process open file table :-
It contains a pointer to the appropriate entry in the system-wide open file table.

Allocation Methods :-

Three major methods of allocating disk space are in wide use :-

(i) Contiguous Allocation :-
It requires that each file occupy a set of contiguous blocks on the disk. It suffers from external fragmentation.

IBM VM/CMS operating system uses contiguous allocation.

(ii) Linked Allocation :-
With linked allocation, each file is a linked list of disk blocks; the disk blocks may be scattered anywhere on the disk. The directory contains a pointer to the first and last blocks of the file.

There is no external fragmentation with linked allocation.

An important variation on linked allocation is the use of a file-allocation table (FAT). This is used by the MS-DOS and OS/2 operating systems. A section of disk at the beginning of each volume is set aside to contain the table. The table has one entry for each disk block and is indexed by block number. The directory entry contains the block number of the first block of the file. The table entry indexed by that block number contains the block number of the next block in the file. This chain continues until the last block, which has a special end-of-file value as the table entry. Unused blocks are indicated by a 0 table value.

(iii) Indexed Allocation :-

Linked allocation solves the external-fragmentation and size-declaration problems of contiguous allocation. However, in the absence of a FAT, linked allocation cannot support efficient direct access, since the pointers to the blocks are scattered with the blocks themselves all over the disk and must be retrieved in order.

Indexed allocation solves this problem by bringing all the pointers together into one location: the index block.

Each file has its own index block, which is an array of disk-block addresses. The $i^{th}$ entry in the index block points to the $i^{th}$ block of the file. The directory contains the address of the index block.

Indexed allocation supports **direct access**, without suffering from external fragmentation. However, it suffers from wasted space.

Since every file must have an index block, so we want the index block to be as small as possible. If the index block is too small, it will not be able to hold enough pointers for a large file, and a mechanism will have to be available to deal with this issue.

Mechanism for this ~~issue~~ purpose include the following :-

(a) Linked Scheme :-

An index block is normally one disk block. Thus it can be read and written directly by itself. To allow for larger files, we can link together several index blocks.

(b) Multilevel Index :-

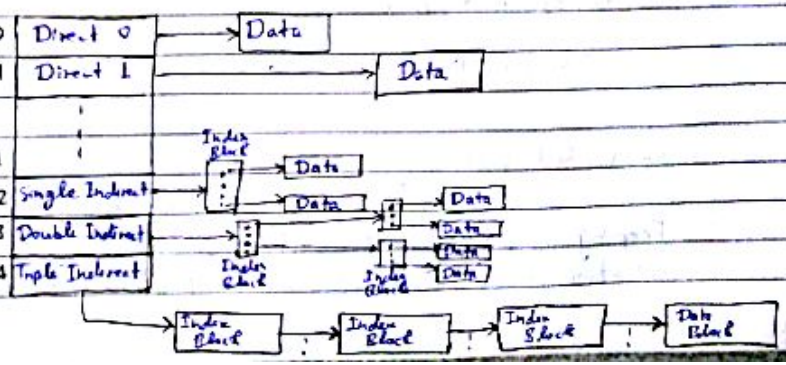
Here we use a first level index block to point to a set of second-level index blocks, which in turn points to the file blocks.

(c) Combined Scheme :-

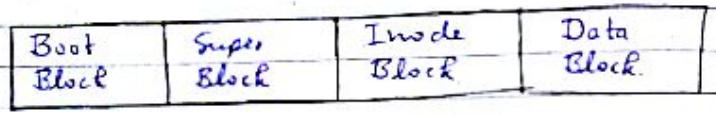
This scheme is used in Unix File System (UFS).

In this scheme, we keep the first 15 pointers of the index block in file's inode. The first 12 of these pointers point to direct blocks; that is they contain addresses of blocks that contain data of the file. Thus, the data for small files do not need a separate index block.

The next three pointers point to indirect blocks. The first points to a single indirect block. The second points to a double indirect block. The last pointer contains the address of a triple indirect block.



## Structure of UNIX File System :-




# If the size of blocks is 1 KB and each address requires 4B, then total memory addressed by each entry of inode table is :-

12 KB + 256 KB + 256×256 KB + 256×256×256 KB

## Free Space Management :-

### (1) Bit Map or Bit Vector :-

The free-space list is implemented as a bit map or bit vector. Each block is represented by 1-bit. If the block is free, the bit is 1, if the block is allocated, the bit is 0.

Example :- Consider a disk where blocks 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26, and 27 are free and the rest of the blocks are allocated. The free space map would be :-

0011110011111100011000000111 00 ...

### (2) Linked List :-

Link together all the free disk blocks, keeping a pointer to the first free block in a special location on the disk and caching it in memory.

## Log-Structured File Systems :-

In these file systems, log-based database recovery techniques are applied to file-system metadata updates. These file systems are also known as log-based transaction-oriented (or journaling) file systems.

These file systems help in keeping the file system consistent.

# Secondary Storage Structure

www.ankurgupta.net

## Auxiliary Memory :-

Devices that provide backup storage are called auxiliary memory. The most common auxiliary memory devices are magnetic disks and tapes.

## Magnetic Disks :-

Each disk platter has a flat circular shape, like a CD. We store information by recording it magnetically on the platters.

The read-write heads are attached to a disk arm that moves all the heads as a unit

The surface of a platter is logically divided into circular tracks, which are subdivided into sectors.

The set of tracks that are at one arm position makes up a cylinder.

## Transfer Rate :-

The transfer rate is the rate at which data flow between the drive and the computer.

## Seek time :-

The time to move the disk arm to the desired cylinder.

## Rotational Latency :-

The time for the desired sector to rotate to the disk head.

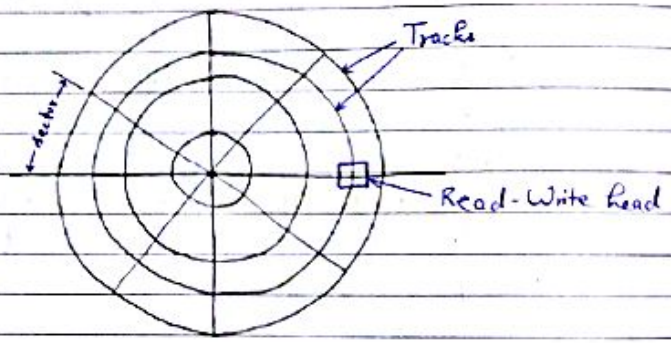
## Transfer time :-

Time required to transfer data to or from the device.

**Access Time :-**

The average time required to reach a storage location in memory and obtain its contents is called the access time.

> Access time = Seek time + Rotational Latency + Transfer time




A disk drive is attached to a computer by a set of wires called an I/O Bus.

The data transfers on a bus are carried out by special electronic processors called controllers :-

(i) **Host Controller :-**
This is the controller at the computer end of the bus.

(ii) **Disk Controller :-**
It is built into each disk drive.

**Addressing in Magnetic Disks :-**

Modern disk drives are addressed as large one-dimensional array of logical blocks.

The one-dimensional array of logical blocks is mapped onto the sectors of the disk sequentially.

Sector 0 is the first sector of the first track on the outermost cylinder. The mapping proceeds in order through that track, then through the rest of the tracks in that cylinder, and then through the rest of the cylinders from outermost to innermost.

**Constant Linear Velocity (CLV) :-**

On media that use constant linear velocity (CLV), the density of bits per track is uniform. The drive increases its rotation speed as the head moves from the outer to the inner tracks to keep the same rate of data moving under the head.

This method is used in CD-ROM and DVD-ROM drives.

**Constant Angular Velocity (CAV) :-**

The disk rotation speed remains constant, and the density of bits decreases from inner tracks to outer tracks to keep the datarate constant.

This method is used in hard disks.

## Disk Scheduling :-

### (i) FCFS Scheduling :-

This algorithm is intrinsically fair, but it generally does not provide the fastest service.

### (ii) Shortest Seek Time First (SSTF) Scheduling :-

The SSTF algorithm selects the request with minimum seek time from the current head position.

It may cause starvation of some requests.

This algorithm is not optimal.

### (iii) SCAN Scheduling :-

The disk arm starts at one end of the disk and moves towards the other end, servicing requests as it reaches each cylinder.

At the other end, the direction of head movement is reversed.

It is sometimes called the elevator algorithm.

### (iv) Circular SCAN (C-SCAN) Scheduling :-

It is similiar to SCAN scheduling, however when the head reaches the other end, it immediately returns to the beginning of the disk, without servicing any request on the return trip.

### (v) LOOK Scheduling :-

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

Versions of SCAN and C-SCAN that follow this pattern are called LOOK and C-LOOK scheduling.

## Disk Formatting :-

A new magnetic disk is a blank slate: It is just a platter of a magnetic recording material.

### (i) Physical Formatting :-

Before a disk can store data, it must be divided into sectors that the disk controller can read and write. This process is called low-level formatting or physical formatting.

It fills the disk with a special data structure for each sector.

### (ii) Logical Formatting :-

Here the operating system stores the initial file-system data structures onto the disk.

These data structures may include maps of free and allocated space and an initial empty directory.

## Raw Disk :-

Some operating systems give special programs the ability to use a disk partition as a large sequential array of logical blocks, without any file-system data structures. This array is sometimes called the raw disk, and I/O to this array is termed as raw I/O.

## Disk Block :-

A disk block is the unit of data transfer b/w disk and memory.

Block size must be multiple of sector size.

Block size can not exceed the size of track

# File Organization and Indexes

## Spanned Records :-

If records can span more than one block in a disk, this organization is called spanned. Whenever a record is larger than a block, we must use a spanned organization.

## Unspanned Records :-

If records are not allowed to cross block boundaries, the organization is called unspanned. This is used with fixed-length records having $B > R$.

## Blocking Factor :-

Suppose that the block size is $B$ bytes. For a file of fixed length records of size $R$ bytes, with $B \geq R$, we can fit $bfr = \lfloor B/R \rfloor$ records per block.

The value bfr is called the blocking factor.

The maximum number of records that can fit in a disk block, is called the blocking factor (bfr).

## Types of Organization of Files :-

There are three types of organization of files :-

### (1) Files of Unordered Records (Heap Files) :-

The records are placed in the file in order in which they are inserted, so new records are inserted at the end of the file.

This organization is often used with additional access paths, such as the secondary indexes.

### (2) Files of Ordered Records (Sorted Files) :-

We can physically order the records of a file on disk based on the values of one of their fields - called the ordering field.

If the ordering field is also a key field of the file, then the field is called the ordering key field.

### (3) Hash Files :-

The idea behind hashing is to provide a function h, called a hash function, that is applied to the hash field value of a record and yields the address of the disk block in which the record is stored.

For most records, we need only a single-block access to retrieve that record.

The search condition must be an equality condition on a single field.

# All the above organizations of files are called primary organization.

## Indexes :-

The index structures typically provide secondary access paths to access the records without affecting the physical placement of records on disk

## Dense and Sparse Indexes :-

A dense index has an index entry for every search key value.

A sparse index has index entries for only some of the search values.

## Types of Indexes :-

### (1) Single Level Ordered Indexes :-

(a) Primary Indexes
(b) Clustering Indexes
(c) Secondary Indexes.

### (2) Multi Level Indexes

## Single Level Ordered Indexes :-

A primary index is specified on the ordering key field of an ordered file of records. If the ordering field is not a key field, then clustering index is used.

A file can have at most one physical ordering field, so it can have at most one primary index or one clustering index, but not both.

A secondary index can be specified on any non-ordering field of a file.

## Primary Indexes :-

A primary index is an ordered file whose records are of fixed length with two fields. The first field is of the same data type as the ordering key field - called the primary key - of the data file, and the second field is a pointer to a disk block.

There is one index entry in the index file for each block in the data file, that has the value of the primary key field for the first record in the block and a pointer to that block.

Total no. of entries in the index is same as the no. of disk blocks in the ordered data file.

A primary index is a non-dense (sparse) index.

A major problem with a primary index - as with any ordered file - is insertion and deletion of records.

## ~~Secondary Indexes~~ Clustering Indexes :-

If records of a file are physically ordered on a non-key field, that field is called the clustering field.

A clustering index is also an ordered file with two fields, the first is same as clustering field, and the second field is a block pointer.

There is one entry in the clustering index for each distinct value of the clustering field, containing the value and the pointer to the first block in the data file that has a record with that value.

A clustering index is also a nondense (sparse) index, because it has an entry for every distinct value of the indexing field, rather than for every record in the file.

## Secondary Indexes :-

It is also an ordered file with two fields. The first field is of the same data type as some non-ordering field of the data file that is an indexing field. The second field is either a block pointer or a record pointer.

There can be many secondary indexes for the same file.

If the secondary index is on a key field, then there is one index entry for each record in the data file. Hence such an index is dense.

If the secondary index is on a non-key field, then numerous records in the data file can have the same value for the indexing ~~file~~ field. There are various options for implementing such an index.

A secondary index provides a logical ordering on the records by the indexing field.

## Multilevel Indexes :-

A multilevel index considers the index file (first level) as an ordered file with a distinct value for each K(i). Hence we create a primary index for the first level.

This index to the first level is called the second level of the multilevel index, and so on.

We require a second level only if the first level needs more than one block of disk storage and so on

Each level reduces the number of entries at the previous level by a factor of $f_o$ - the index fan-out.

If the first level has $r_1$ entries, and the blocking factor - which is also the fan-out - for the index is $bf_i = f_o$, then the first level needs $\lceil (r_1/f_o) \rceil$ blocks, which is therefore the number of entries, $r_2$, needed at the second level of the index.

## B-Trees :-

The B-Tree is a search tree with additional constraints that ensure that the tree is always balanced.

A B-tree of order p, when used as an access structure on a key field to search for records in a data file, can be defined as follows:-

(1) Each internal node in the B-tree is of the form :-

$<P_1, <K_1, Pr_1>, P_2, <K_2, Pr_2>, \ldots <K_{q-1}, Pr_{q-1}>, P_q>$

where $q \leq p$. Each $P_i$ is a tree pointer - a pointer to another node in the B-tree. Each $Pr_i$ is a data pointer - a pointer to the record whose search key field value is equal to $K_i$.

(2) Within each node, $K_1 < K_2 < \ldots < K_{q-1}$

(3) For all search key field values X in the subtree pointed at by $P_i$, we have :-

$K_{i-1} < X < K_i$ for $1 < i < q$,

$X < K_i$ for $i = 1$, and

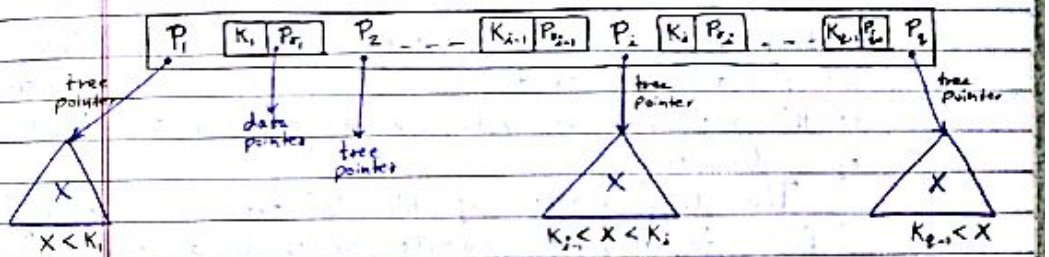
$K_{i-1} < X$ for $i = q$.

(4) Each node has at most p tree pointers.

(5) Each node, except the root and leaf nodes, has at least $\lceil (p/2) \rceil$ tree pointers. The root node has at least two tree pointers unless it is only node in the tree.

(6) A node with q tree pointers, $q \leq p$, has $q-1$ search key field values.

(7) All leaf nodes are at the same level. Leaf nodes have the same structure as internal nodes except that all of their tree pointers $P_i$ are null.




(A node in a B-tree with q-1 search values)

Example :- Suppose the search field is V = 9 bytes long, the disk block size is B = 512 bytes, a record (data) pointer is $P_r$ = 7 bytes, and a block pointer is P = 6 bytes. then :-

$$(p * P) + ((p-1) * (P_r + V)) \leq B$$

$$\Rightarrow (p * 6) + (p-1) * 16 \leq 512$$

$$\Rightarrow 22 * p \leq 528$$

$$\Rightarrow p \leq 24.$$

## $B^+$-Trees :-

In a $B^+$-tree, data pointers are stored only at the leaf nodes on the tree; hence, structure of the leaf nodes differ from the structure of the internal nodes.

The leaf nodes have an entry for every value of the search field, along with a data pointer to the record if the search field is a key field. For a nonkey search field, the pointer points to a block containing pointers to the data file records, creating an extra level of indirection.

The leaf nodes of the $B^+$ tree are usually linked together to provide ordered access on the search field to the records. Some search field values from the leaf nodes are repeated in the internal nodes of the $B^+$ tree to guide the search.

The structure of the internal nodes of a $B^+$-tree of order $p$ is as follows :-

(1) Each internal node is of the form :-

$< P_1, K_1, P_2, K_2, \ldots P_{q-1}, K_{q-1}, P_q >$

where $q \leq p$ and each $P_i$ is a tree pointer.

(2) Within each internal node, $K_1 < K_2 < \ldots < K_{q-1}$

(3) Each internal node has at most $p$ tree pointers.

(4) For all search field values $X$ in the subtree pointed at by $P_i$, we have :-

$K_{i-1} < X \leq K_i$ for $1 < i < q$,

$X \leq K_i$ for $i = 1$, and

$K_{i-1} < X$ for $i = q$.

(5) Each internal node, except the root, has at least $\lceil (p/2) \rceil$ tree pointers. The root node has at least two tree pointers if it is an internal node.

(6) An internal node with $q$ pointers, $q \leq p$, has $q-1$ search field values.

The structure of the leaf nodes of a $B^+$ tree of order $p$ is as follows:-

(1) Each leaf node is of the form :-

$< < K_1, P_{r_1} >, < K_2, P_{r_2} >, \ldots < K_{q-1}, P_{r_{q-1}} >, P_{next} >$

where $q \leq p$, each $P_{r_i}$ is a data pointer, and $P_{next}$ points to the next leaf node of the $B^+$-tree.

(2) Within each leaf node, $K_1 < K_2 < \ldots < K_{q-1}$, $q \leq p$.

(3) Each $P_{r_i}$ is a data pointer that points to the record whose search field value is $K_i$.

(4) Each leaf node has at least $\lfloor (p-1)/2 \rfloor$ values.

(5) All leaf nodes are at the same level.

†† $P_{next}$ pointer provides ordered access to the data records on the indexing field.

Because entries in the internal nodes of a $B^+$-tree include search values and tree pointers without any data pointers, more entries can be packed into an internal node of a $B^+$ tree than for a similiar B-tree.

Because the structures for internal nodes and for leaf nodes of a $B^+$ tree are different, the order $p$ can be different.

We use $p$ to denote the order for internal nodes and $p_{leaf}$ to denote the order for leaf nodes.

Example:- For a $B^+$-tree, the search key field is $V = 9$ bytes long, the block size is $B = 512$ bytes, a record pointer is $P_r = 7$ bytes, and a block pointer is $P = 6$ bytes. Calculate the order of internal and leaf nodes.

Solution:- For internal node :-

$$(p * P) + (p-1) * V \leq B$$

$$\Rightarrow 6p + 9 \times (p-1) \leq 512$$

$$\Rightarrow 15p \leq 521$$

$$\Rightarrow p \leq 34$$

$$\Rightarrow p = 34 \text{ Ans}$$

For leaf nodes :-

$$p_{leaf} \times (P_r + V) + P \leq B$$

$$\Rightarrow (7+9) \times p_{leaf} + 6 \leq 512$$

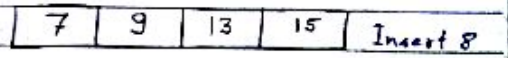
$$\Rightarrow 16 \times p_{leaf} \leq 506$$

$$\Rightarrow p_{leaf} = 31 \text{ Ans}$$

Insertion in $B^+$ trees :-

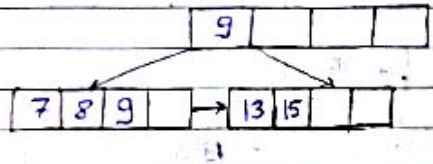
Overflow :-

When number of search-key values exceed $p-1$.



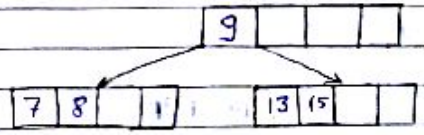

Splitting Leaf Node into two nodes :-

→ 1st node contains $\lceil p/2 \rceil$ values.
→ 2nd node contains remaining values.
→ Copy the maximum search-key value of the 1st node to the parent node.



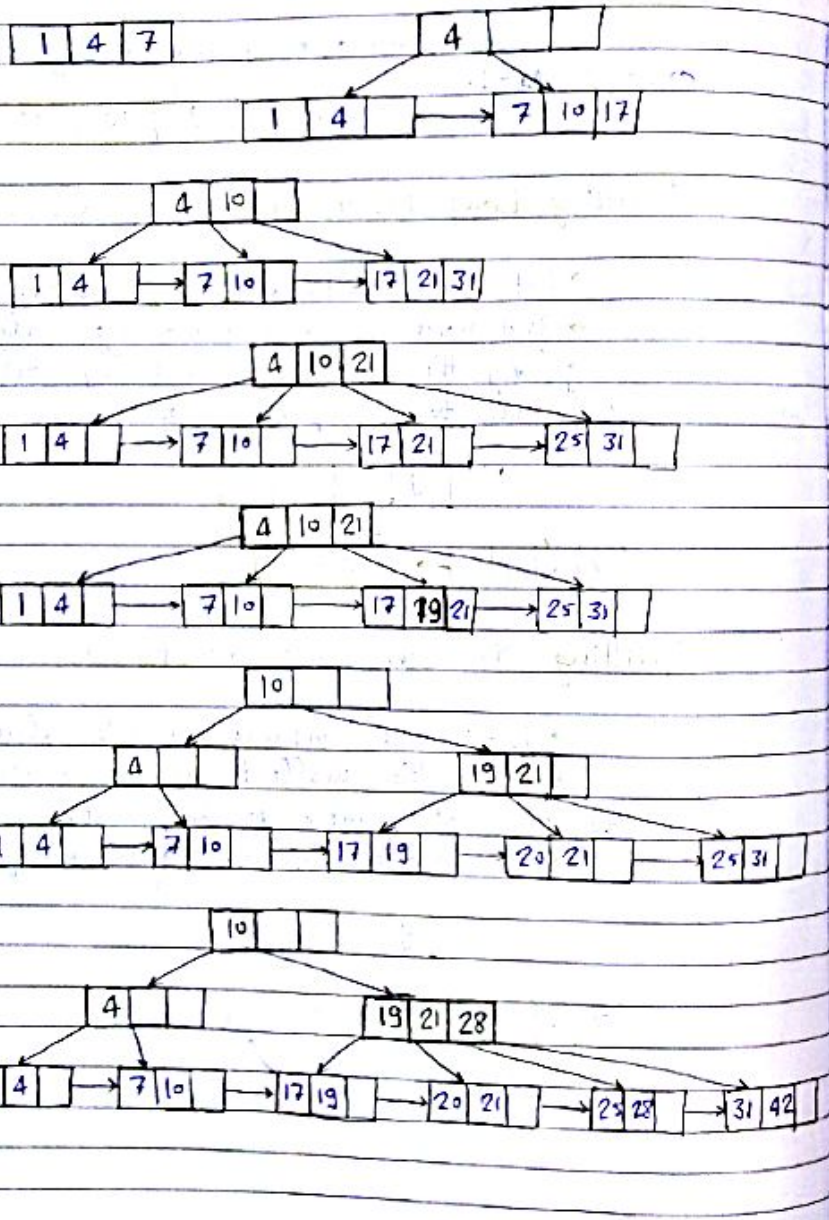

Splitting Non-Leaf node into two nodes :-

→ 1st node contains $\lfloor (p-1)/2 \rfloor$ values.
→ Move the smallest of the remaining values, together with pointer, to the parent.
→ 2nd node contains the remaining values.



Example :- Construct a $B^+$ tree for (1, 4, 7, 10, 17, 21, 31, 25, 19, 20, 28, 42) with $p = 4$.

Solution :-




Deletion in $B^+$ tree :-

Underflow :-

When number of search-key values < $\lceil (p-1)/2 \rceil$

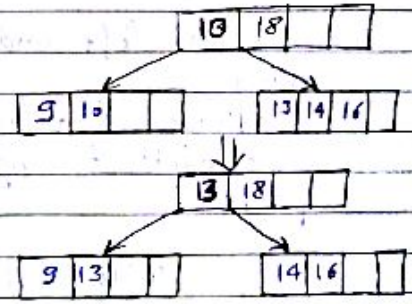

(i) Leaf Node :-

(a) Redistribute to sibling

→ Right node should not be less than left node.

→ Move the smallest value of the right node to left node.

→ Replace the between value in parent by the smallest value of the right node.

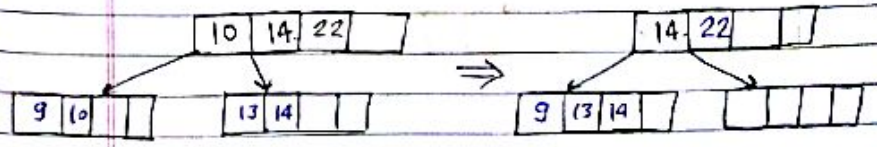



(b) Merge :-

→ Both left and right nodes have $\lfloor (p-1)/2 \rfloor$ values.

→ Move all values, pointers to left node.

→ Remove the between value in parent.

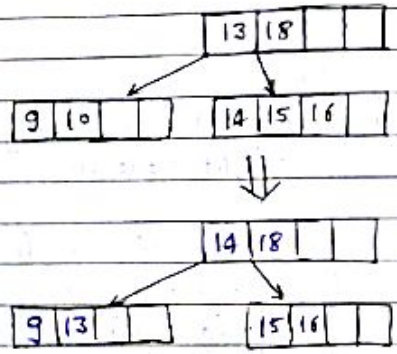


(ii) Non-Leaf Node :-

(a) Redistribute to sibling :-

→ Through parent

→ Right node should not be less than left node

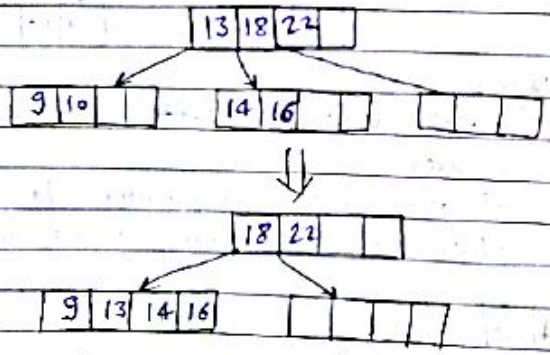



(b) Merge :-

→ Both left and right node have $\lfloor (p-1)/2 \rfloor$ values.

→ Move all values, pointers to left node.

→ Delete the right node, and pointers in parent.




Example :- Delete 28, 31, 21, 25, 19 from the given B+ tree.

Solution :-