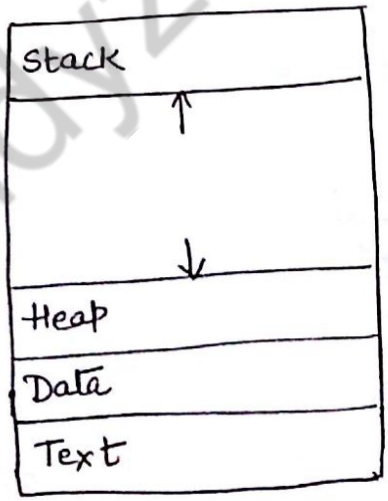
PROCESS CONCEPT UNIT-3

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 entry which represents the basic unit of work to be implemented in the system".

To put in simple words - when a program is loaded into the memory and it becomes a process, it can be divided into four sections - stack, heap, text and data. The following image shows a simplified layout of a process inside main memory.



Stack:- The process stack contains the temperory data such as method/function parameters, return address and local variables.

Heap:- This is dynamically allocated memory to a process during its runtime.

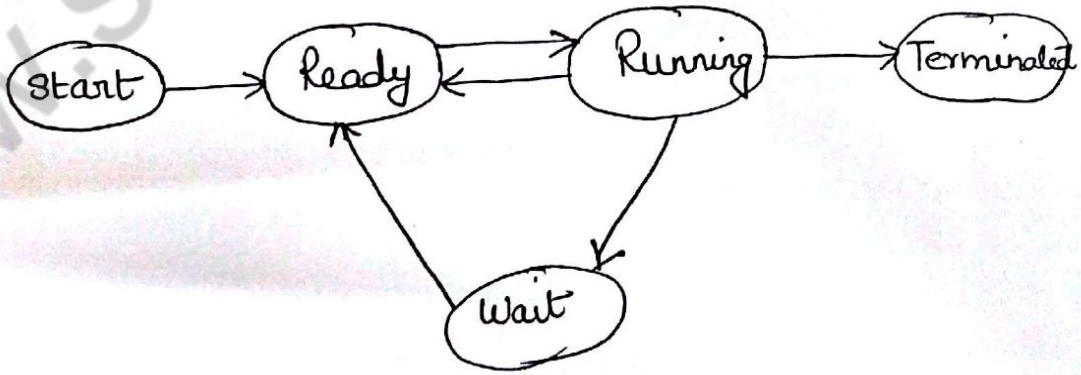
Text:- This includes the current activity represented by the value of Program Counter and the contents of the processor's registers.

Data:- This section contains the global & static variables.

Program:- A computer program is a collection of instructions that perform a specific task when executed by a computer. "A process is a dynamic instance of a program".

A part of a computer program that performs a well defined task is known as an algorithm. A collection of computer programs, libraries & related data are referred to as a software.

Process Life Cycle :-



Process States

When a process executes, it passes through diff states. These stages may differ in different OS.

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

Ready :- The process is waiting to be assigned to the processor. Ready processes are waiting to have the processor allocated to them by the O.S. so that they can run. Processes may come to this state after "Start" state or while running it by but interrupt by the scheduler to assign CPU to some other processes.

Running :- Once the process has beenassigned to a processor by the OS scheduler, the process state is set to running and the processor executes its instructions.

Waiting :-

Processes move into 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 :- Once the process finishes its exe", or it is terminated by the O.S., it is moved to the terminated state where it waits to be removed from main memory.

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 information needed to keep track of a process as listed below in table :-

1. Process State :- The current state of the process ie. whether it is ready, running, waiting or whatever.
2. Process Privileges :- This is req. to allow/disallow access to system resources.
3. Process ID :- Unique identification for each of the process in the O.S.
4. Pointer :- A pointer to parent process.
5. Program Counter :- is a pointer to the address of the next instruction to be executed for this process.
6. CPU registers :- Various CPU registers where process needs to be stored for exec" for running state.
7. CPU scheduling information :- Process priority & other scheduling

Process ID
State
Pointer
Priority
Program Counter
CPU registers
I/O Information
Accounting Informat"
:
etc :

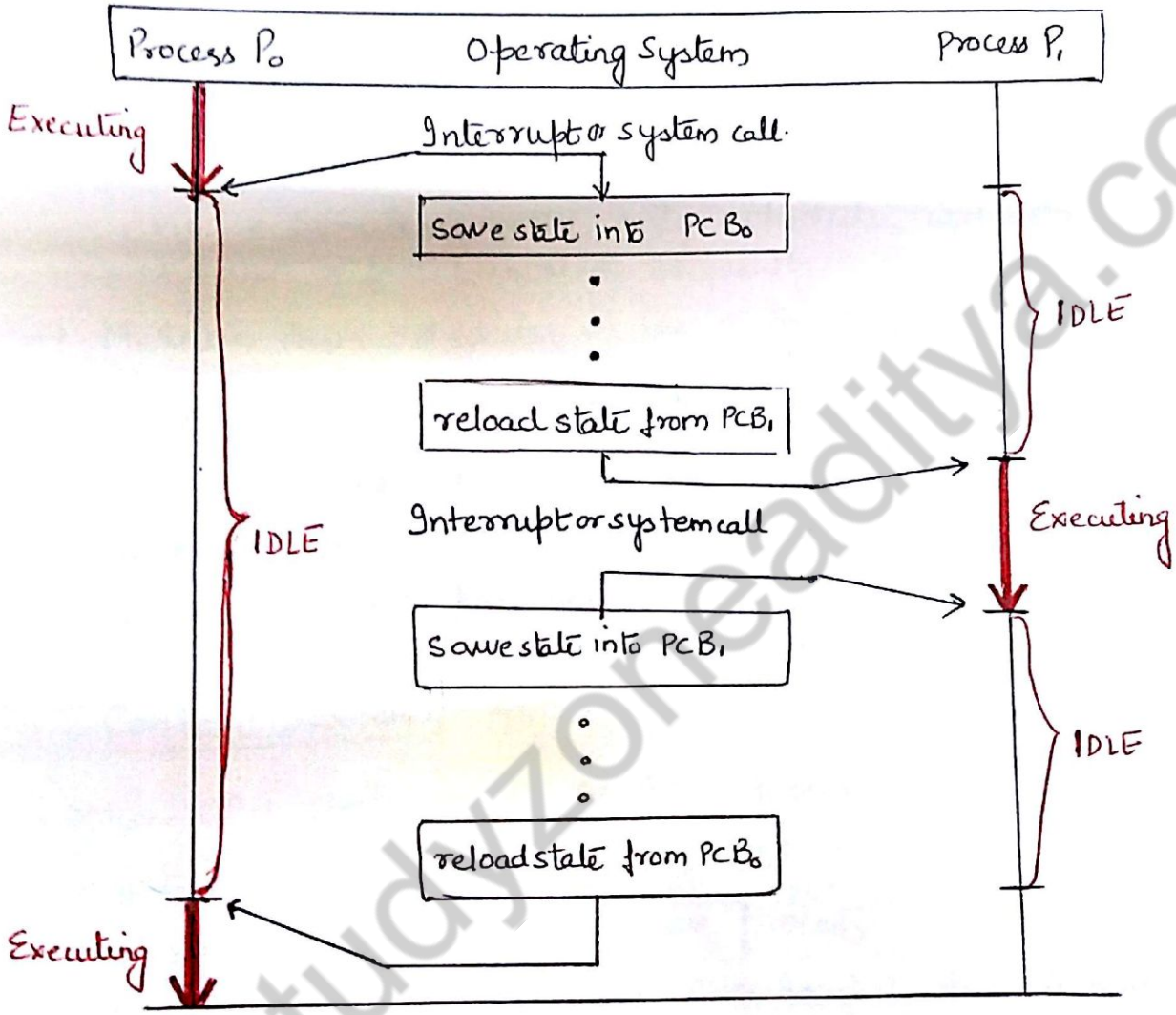
Simplified Diag of PCB

8. Memory Management information :- This includes the information of page table, memory limits, segment table depending on memory used by the OS.

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

10. I/O status information :- This includes a list of I/O devices allocated to the process, list of open files & so on.

CPU Switch From Process to Process.



Schedulers

→ Long term scheduler :- or (job scheduler) - selects which processes should be brought into the ready queue.
→ must be slow (invoked less frequently).

→ Short term scheduler :- or (CPU scheduler) - selects which process should be executed next and allocates CPU. • Sometimes the only scheduler in the system.
→ must be fast (invoked frequently).

→ Medium term scheduler :- is a part of swapping. It removes the process from the memory. It reduces the degree of multiprogramming. The medium-term scheduler is in charge of handling the swapped out-processes.

Context Switch

→ When CPU switches to another process, the system must save the state of the old process and load the saved state for the new process.

→ Context-Switch time is overhead; the system does no useful work while switching.

→ Time dependent on hardware support.

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 O.S. Such OS systems allow more than one process to be loaded into the executable memory at a time and the loaded process shares the CPU using multiplexing.

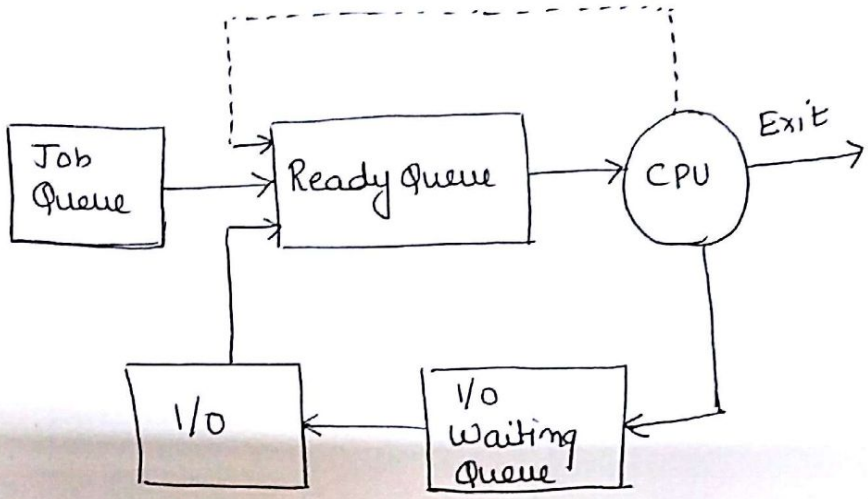
Process Scheduling Queues

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 & move to its new state.

→ **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 process



Process Scheduling Queues.

S.No	Long Term Scheduler	Short-Term Scheduler	Medium-Term Scheduler
1.	It is a job scheduler	It is a CPU scheduler	It is a process swapping scheduler
2.	Speed is lesser than S.T. scheduler	speed is fastest among other two	speed is in b/w both short & long term scheduler
3.	It controls the degree of multi-programming	It provides lesser control over degree of multiprogramming	It reduces the degree of multiprog-ramming.
4.	It is almost absent or minimal in time sharing system	It is also minimal in time sharing system	It is a part of time sharing systems
5.	It selects processes from pool & loads them into memory for execution	It select those processes which are ready to execute	It can re-introduce the process into memory and execut^n can be continued.

Scheduling Criteria

Different CPU-scheduling algorithms have different properties and may favour one class of process or another.

Many criteria have been suggested for comparing CPU scheduling algorithms. The characteristics used for comparison can be make a substantial difference in the determination of best algorithm.

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

Throughput :- If the CPU is busy executing processes, then work is being done. One measure of work is the number of processes completed per time unit called throughput. For long processes, this rate may be 1 process per hour for short transactions, throughput might be 10 processes per second.

Turnaround time :- From the point of view of a particular process, the important criterion is how long it takes to execute that process.

The interval from the time of submission of a process to the time of completion is the turn around time.

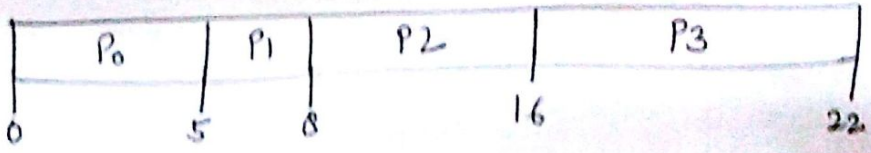
Process Scheduling Algorithms

A process Scheduler schedules different processes to be assigned to the CPU based on particular scheduling algorithms. These algorithms are either preemptive or non-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.

1. First Come First Serve (FCFS)

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

Process	Arrival time	Execute Time	Service Time
P_0	0	5	0
P_1	1	3	5
P_2	2	8	8
P_3	3	6	16



Wait time for each process is as follows:-

Process	Wait Time : Service time - Arrival time
P_0	0 - 0 = 0
P_1	5 - 1 = 4
P_2	8 - 2 = 6
P_3	16 - 3 = 13

Average W.T = (0+4+6+13)/4 = 5.75

2. Shortest Job Next:- (SJN).

→ This is also known as shortest job first or SJF

→ This is a non-preemptive scheduling algo.

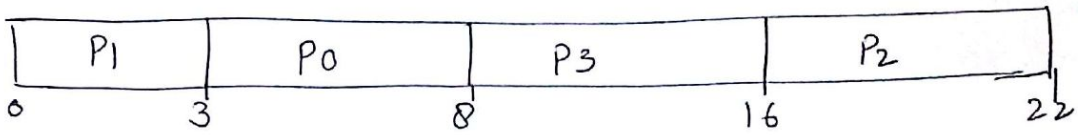
→ Best approach to minimize waiting time.

→ 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 processor should know in advance how much time process will take.

Process	Arrival time	Execut^n Time	Service time
P_0	0	5	3
P_1	1	3	0
P_2	2	8	16
P_3	3	6	8



Wait time for each process –

P_0 – 3-0 = 3

P_1 – 0-0 = 0

P_2 – 16-2 = 14

P_3 – 8-3 = 5

Average W.T. = (3+0+14+5) /4 = 5.50

3. Priority Based Scheduling :-

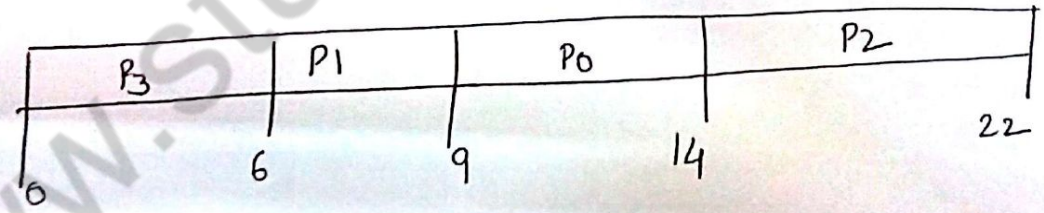
→ Priority scheduling is a non-preemptive algorithm and one of the most common scheduling algorithm in batch systems.

→ 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 FCFS basis.

→ Priority can be decided based on memory requirements, time requirements or any other resource requirement

Process	Arrival Time	Execut^n Time	Priority	Service time
P_0	0	5	1	9
P_1	1	3	2	6
P_2	2	8	1	14
P_3	3	6	2	0



Waiting time :- P_0 – 9-0 = 9

P_1 – 6-1 = 5

P_2 – 14-2 = 12

P_3 – 6-0 = 6

A.W.T. = (9+5+12+0)/4 = 6.5

4. Shortest Remaining Time : (SRT).

→ 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.

5. Round Robin Scheduling :-

→ Round robin is the preemptive process scheduling algorithm.

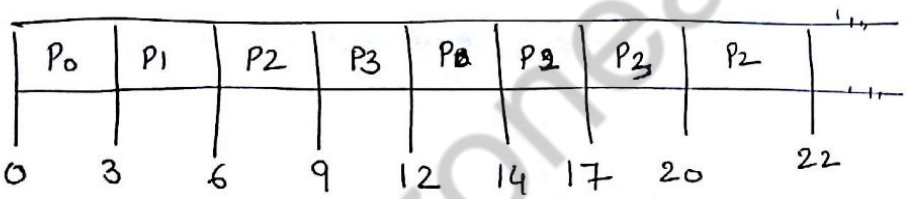
→ Each process is provided a fix time to execute, it is called a quantum.

→ 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.

–



Waiting time :- P_0 – (0-0)+(12-3) = 9

P_1 – (3-1) = 2

P_2 – (6-2)+(14-9)+(20-17) = 12

P_3 – (9-3)+(17-12) = 11

Average W.T. = (9+2+12+11)/04 = 8.5.

6 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 eg. 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.

Thread :-

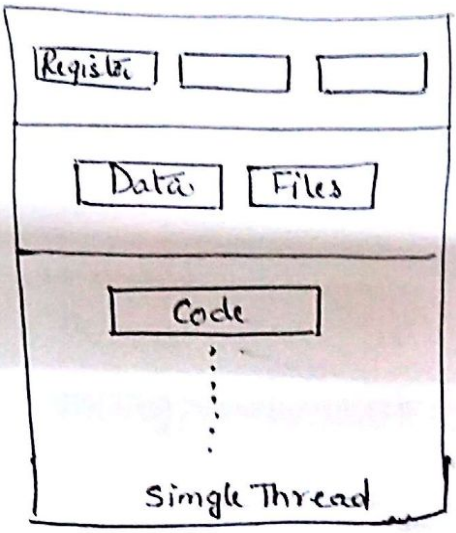
A thread is a flow of execution through the process code, with its own program counter that keeps track of which instruction to execute next, system registers which hold its current working variables, and a stack which contains the execution history.

A thread shares with its peer threads few information like code segment, data segment & open files. When one thread alters a code segment memory item, all other threads see that.

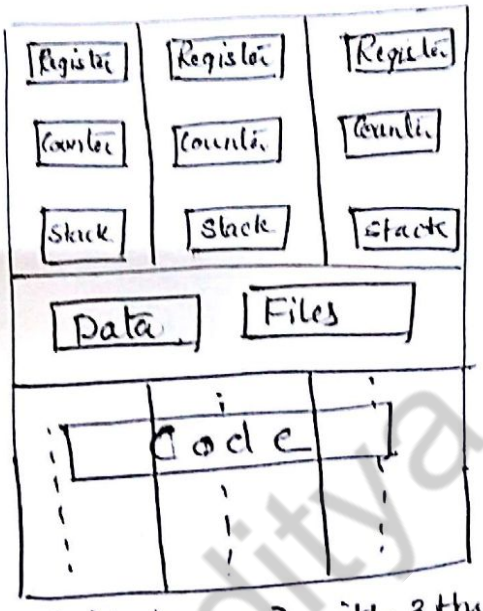
A thread is also called a light weight process. Thread provide a way to improve application performance through parallelism. Thread represents a s/w approach to improving performance of O.S. by reducing the overhead thread is equivalent to a classical process.

Each thread belongs to exactly one process and no thread can exist outside a process. Each thread represents a separate flow of control. Threads have been successfully used in implementing n/w servers & web server. They also provide a suitable foundation for parallel execution of application on shared memory multiprocessors.

The following figure shows the working of a single-threaded & a multithreaded process.



Single Process P with Single thread



Single-process P with 3 threads.

	PROCESS	THREAD
1.	Process is heavyweight & resource intensive.	- Thread is light wt, taking lesser resources than a process.
2.	Process switching needs interact" with OS.	- Thread switching does not need to interact with OS.
3.	In multiprocessing environments each process executes the same code but has its own memory & file resources.	- All threads can share same set of open files, child processes.
4.	If one process is blocked, then another process can execute until the first process is unblocked	- While one thread is blocked & waiting, a second thread in the same task can run.
5.	Multiple process without using threads use more resources	- Multiple threaded processes use fewer resources.
6.	In multiple process each process operates independently of the others.	- One thread can read, write or change another thread's data.

Advantages of threads:

→ Threads minimize the context switching time.
→ Use of threads provide concurrency within a process.
→ Efficient comm"
→ It is more economical to create & context switch threads.
→ Threads allow utilization of multiprocessor architectures to a greater scale & efficiency.

Types of threads :-

Threads are implemented in following 2 ways:-

→ User level threads :- User managed threads
→ Kernel level threads :- OS manage threads acting on kernel, an O.S. core.

Deadlocks

System Model:- A system consists of finite number of resources to be distributed among a number of competing processes.

→ Resources types :- $R_1, R_2 \cdots R_m$

→ Each resource type R_i has 1 or more instance.

→ Each process utilizes a resource as follows:-

- → Request
- → Use
- → Release.

The Deadlock Problem

→ A deadlock consist of a set of blocked processes, each holding a resource and waiting to acquire a resource held by another process in the set.

Eg.
- A system has 2 disk drives.
- P_1 and P_2 each hold one disk drive & each needs the one

Eg
- Semaphores A and B, initialized to 1.

P_0	P_1
wait(A);	wait(B);
wait(B);	wait(A);

Deadlocks

System Model:- A system consists of finite number of resources to be distributed among a number of competing processes.

→ Resources types :- $R_1, R_2 \cdots R_m$

→ Each resource type R_i has 1 or more instance.

→ Each process utilizes a resource as follows:-

- → Request
- → Use
- → Release.

The Deadlock Problem

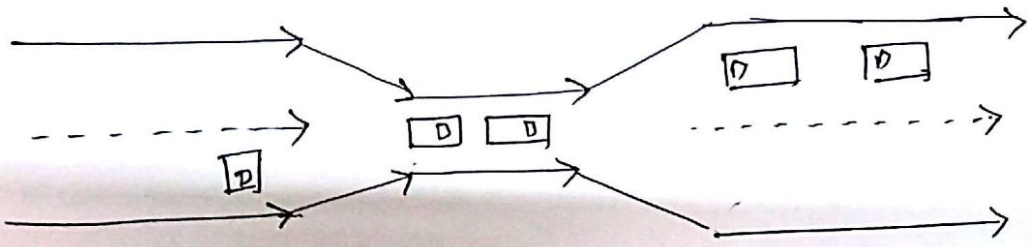
→ A deadlock consist of a set of blocked processes, each holding a resource and waiting to acquire a resource held by another process in the set.

Eg.
- A system has 2 disk drives.
- P_1 and P_2 each hold one disk drive & each needs the one

Eg
- Semaphores A and B, initialized to 1.

P_0	P_1
wait(A);	wait(B);
wait(B);	wait(A);

Bridge Crossing Example



→ Traffic only in one direction
→ The resource is one one lane bridge
→ If a deadlock occurs, it can be resolved if one car backs up (preempt resources & rollback)
→ Several cars may have to be backed up if a deadlock occurs.
→ Starvation is possible.

Deadlock Characterization

Deadlock can arise if 4 conditions hold simultaneously

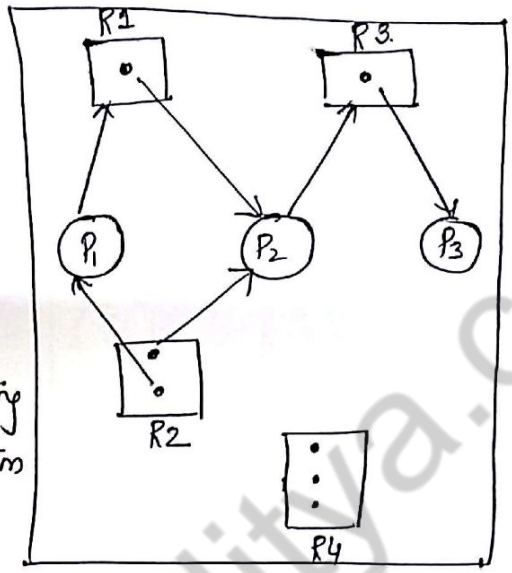
1. Mutual Exclusion:- only one process at a time can use the resource.
2. Hold & wait :- a process holding atleast one resource is waiting to acquire additional resources held by other processes.
3. No preemption :- a resource can be released only voluntarily by the process holding it after that process has completed its task.
4. Circular Wait :- there exists a set $\{P_0, P_1, \ldots, P_0\}$ of waiting processes such that P_0 is waiting for a resource that is held by P_1, P_1 is waiting for a resource that is held by $P_2 \ldots P_{n-1}$ is waiting for a resource that is held by P_n & P_n is waiting for a resource that is held by P_0.

Resource Allocation Graph

A set of vertices V and a set of edges E.

→ V is partitioned into 2 types:

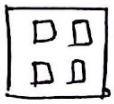
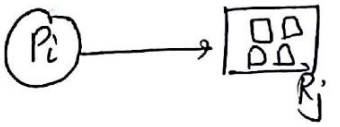
- P = {P1, P2, P3 ... Pn}, the set consisting of all the processes in the system.
- R = {R1 R2 .. Rm} the set consisting of all the resources in the system

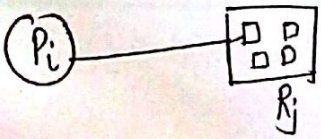


→ Request edge - directed graph.

$P_i \rightarrow R_j$

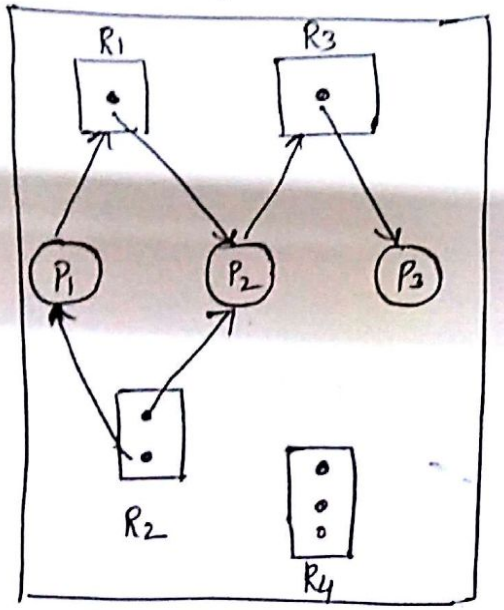
→ Assignment edge - directed edge $R_j \rightarrow P_i$.

- Process — ○
- Resource type with instances 
- P_i request instance of R_j 

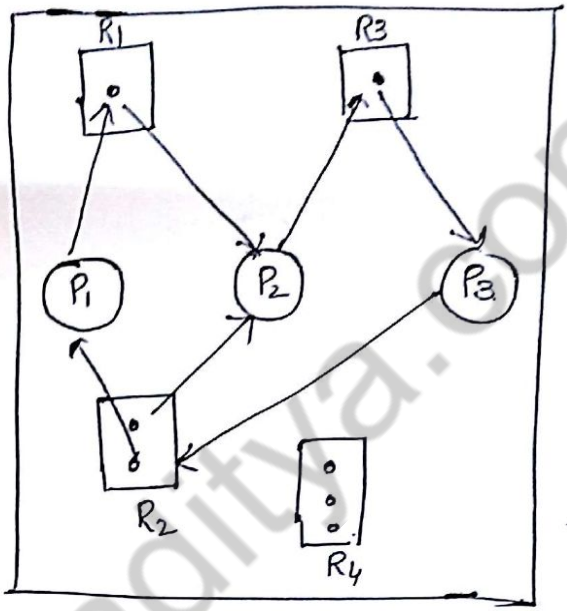
- P_i is holding an instance of R_j 

Resource Allocation Graph with a Deadlock

Before P_3 requested an instance of R_2

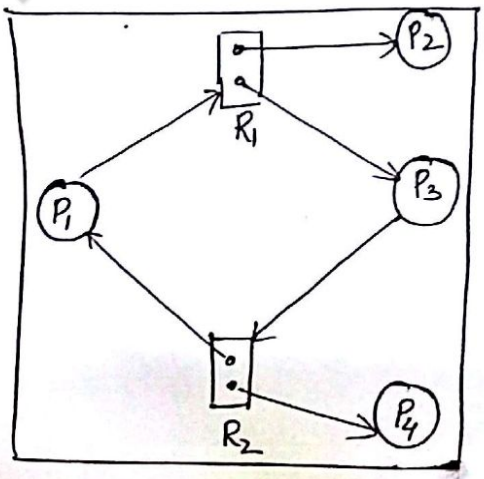


After P_3 requested an instance of R_2



Graph with a Cycle but not a deadlock

Process P_4 may release its instance of resource type R_2. That resource can then be allocated to P_3, thereby breaking the cycle.



→ If a resource allocation graph contains no cycles ⇒ no deadlock.

→ If a resource allocation graph contains a cycle and if only one instance exists per resource type ⇒ deadlock.

→ If a resource allocation graph contains a cycle and if several instances exists per resource type ⇒ possibility of deadlock.

Methods of handling Deadlocks

→ Prevention :- Ensure that the system will never enter a deadlock state.

→ Avoidance :- Ensure that the system will never enter an unsafe state.

→ Detection :- Allow the system to enter a deadlock state and then recover.

→ Do nothing :- Ignore the problem and let the user or system administrator respond to the problem used by most O.S, including Windows and UNIX.

Deadlock Prevention :-

To prevent deadlock, we can restrain the ways that a request can be made :-

→ Mutual Exclusion :- The mutual-exclusion condition must hold for non-sharable resources.

→ Hold and Wait :- we must guarantee that whenever a process requests a resource, it does not hold any other resources.

- Require a process to request and be allocated all its resources before it begins execution, or allow a process to request resources only when the process has none.
- Result :- Low resource utilization, starvation possible

→ No preemption :-

- If a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released.
- Preempted resources are added to the list of resources for which the process is waiting.
- A process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting.

→ Circular Wait :- impose a total ordering of all resources types, and require that each process requests resources in an increasing order of enumeration.

For eg.

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

Deadlock Avoidance

Requires that the system has some additional 'a priori' information available:-

→ Simplest and most useful model requires that each processes declare the maximum number of resources of each type that it may need.

→ The deadlock avoidance algorithm dynamically examines the resource allocation state to ensure that there can never be a circular-wait condition.

→ A resource allocation state is defined by the number of available & allocated resources, & the maximum demands of the processes.

Safe State:-

→ When a process requests an available resource, the system must decide if immediate allocation leaves the system in a safe state.

→ A system is in a safe state only if there exists a safe sequence.

→ A sequence of processes $< P_1, P_2, \ldots, P_n >$ 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 currently available resources plus resources held by all P_j, with $j < i$.

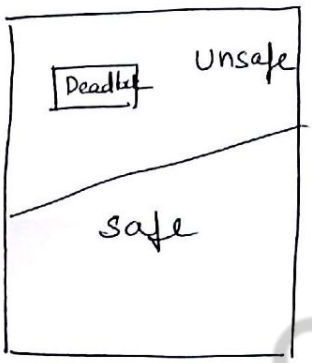
That is:-

- If the P_i resource needs are not immediately available, then P_i can wait until all P_j have finished
- When P_j is finished, P_i can obtain needed resources, execute, return allocated resources & terminate
- When P_i terminates, P_{i+1} can obtain its needed resources & so on.

→ If a system is in safe state : no deadlock.

→ If a system is in unsafe state ⇒ possibility of deadlock.

→ Avoidance ⇒ ensures that a system will never enter an unsafe state.



Avoidance Algorithms

→ For a single instance of a resource type, use a resource allocation graph.

→ For multiple instances of a resource type, use the banker's algorithm.

Resource Allocation Graph scheme :-

① → Introduce a new kind of edge called a claim edge.

② → Claim edge $P_i \dashrightarrow R_j$ indicates that process P_j may request resource R_j which is represented by a dashed line.

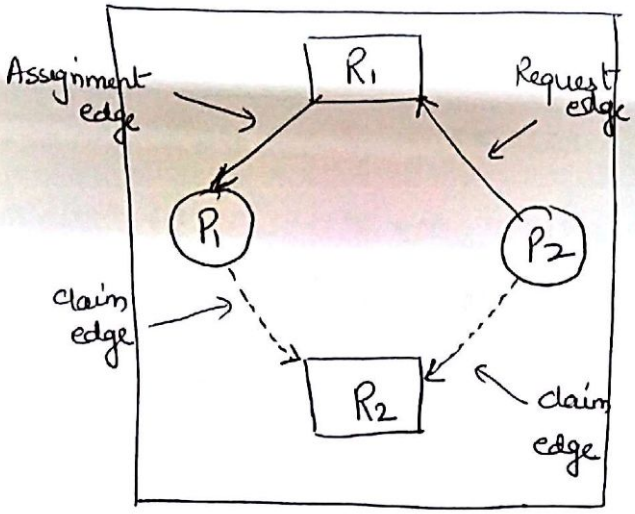
③ → A claim edge converts to an ~~assignment~~ request edge when the resource is allocated to the process.

⑤ → When a resource is released by a process, an assignment edge reconverts to a claim edge.

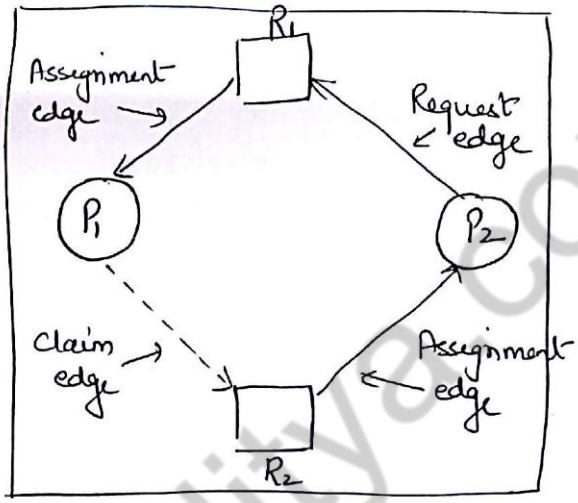
⑥ → Resources must be claimed a priori in the system.

④ → A request edge converts to an assignment edge when the resource is allocated to the process.

Resource allocation graph with claim edges



Unsafe state in Resource allocation graph.



Resource Allocation Graph Algorithm

→ Suppose that process P_i requests resource R_j

→ The request can be granted only if converting the request edge to an assignment edge does not result in the formation of a cycle in the resource allocat^n graph.

Banker's Algorithm

→ Used when there exists multiple instances of a resource type.

→ Each process must a priori claim maximum use.

→ When a process requests a resource, it may have to wait.

→ When a process gets all its resources, it must return them in a finite amount of time.

Data Structures for the Banker's algorithm

Let n = number of processes, and m = number of resource types.

→ Available :- Vector of length m. If available[j]=k, there are k instances of resource type R_j available.

→ Max :- n×m matrix. If Max[i,j]=k, then processes P_i may request at most k instance of resource type R_j.

→ Allocation :- n×m matrix. If Allocation[i,j]=k then P_i is currently allocated k instances of R_j.

→ Need :- n×m matrix. If Need[i,j]=k, then P_i may need k more instances of R_j to complete its task.

$$Need[i,j] = Max[i,j] - Allocation[i,j].$$

Deadlock Detection

Deadlock Detection

→ For deadlock detection, the system must provide
- An algorithm that examines the state of the system to detect whether a deadlock has occcured.
- An algorithm to recover from the deadlock.

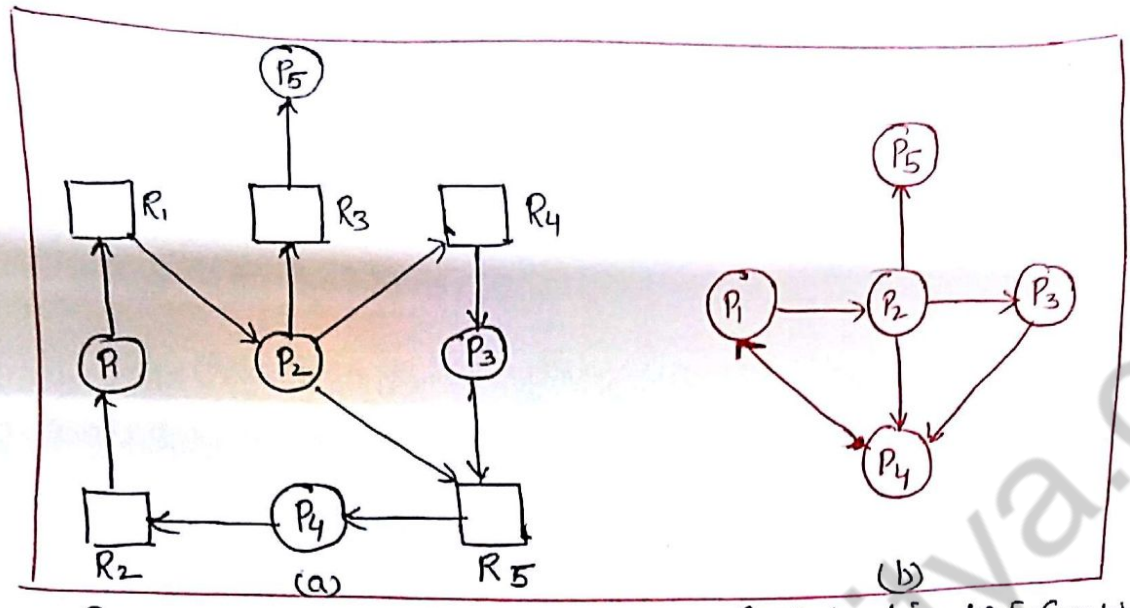
→ A detection-and-recovery scheme requires various kinds of overhead.
- Run-time costs of maintaining necessary information and executing the detection algorithm.
- Potential losses inherent in recovering from a deadlock.

Single instances of each resource type

→ Requires the creation & maintenance of a wait for graph.
- Consists of a variant of the resource allocat^n graph.
- The graph is obtained by removing the resource nodes from a resource allocation graph & collasping the appropriate edges.
- Consequently all nodes are processes.
- $P_i \rightarrow P_j$ if P_i is waiting for P_j.

→ Periodically invoke an algorithm that searches for a cycle in the graph.
- If there is a cycle, there exists a deadlock.
- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph.

Resource allocation Graph & Wait for Graph



Resource allocation Graph

Corresponding W-F-Graph

Multiple instances for a Resource Type

Required data structures

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

→ **Allocation** :- An nxm matrix defines the number of resources of each type currently allocated to each process.

→ **Request** :- A nxm matrix indicates the current request of each process. If Request[ij] = k, then process Pi is requesting k more instances of resource type Rj.

Detection Algorithm Usage

→ When, and how often, to invoke the detection algo depends on:
- How often is a deadlock likely to occur?
- How many processes will be affected by deadlock when it happens?

→ If the detection algorithm is invoked arbitrary, there may be many cycles in the resource graph & so we would not be able to tell which one of the many deadlocked processes "caused" the deadlock.

→ If the detection algorithm is invoked for every resource request, such an action will be incur a considerable overhead in compulation time

→ If the detection algorithm is invoked for every resource request, such an action will incurs a considerable overehead in compulation.

→ A less expensive alternate is to invoke the algorithm when CPU utilization drops below 40% for eg.
- This is based on the observation that a deadlock eventually cripples system throughput and causes CPU utilization to drop.

Recovery From Deadlock

Two approaches

- Process termination
- Resource preemption

Process Termination

→ Abort all deadlocked process.

- This approach will break the deadlock, but at great expense.

→ Abort one process at a time untill the deadlock cycle is eliminated.

- This approach incurs considerable overhead since, after each process is aborted, a deadlock-detection algorithm must be re-invoked to determine whether any processes are still deadlocked.

→ Many factors may affect which process is chosen for termination.

- What is the priority of the process?
- How long has the process run so far & how much longer will the process need to run before completing its task.
- How many and what type of resources has the process used?
- How many resources does the process need in order to finish its tasks?
- How many processes will need to be terminated?
- Is the process interactive or batch?

Resource Preemption

→ With this approach, we successfully preempt some resources from processes & give these resources to other processes until the deadlock cycle is broken.

→ When preemption is required to deal with deadlocks, then three issues need to be addressed :-

- Selecting a victim :- which resources and which processes are to be preempted.
- Rollback :- If we preempt a resource from a process, what should be done with that process?
- Starvation :- How do ensure that starvation will not occur? That is, how can we guarantee that resources will not always be preempted from the same process?