"Operating System"

__/__/____

'Deadlock'

* In multi-Programming environment, we have several Processes competing for finite number of resources
* Process requests a Resource (R), if R is not available (taken by other Process), process enters in a waiting state. Sometimes that waiting Process is never able to change its state because the resource, it has requested is busy (forever) called 'DEADLOCK (DL)'.

↓

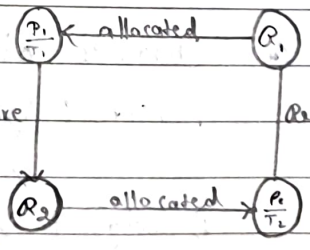
∴ 2 or more Processes are waiting on some resource's availability, which will never be available as it is also busy with some other Process -> This is said to be in 'DL'.

* DL → it is a bug present in Process/thread Synchronization model
* In DL → Processes never finish Executing and the System resources are tied up, preventing other jobs from starting

⊙ Example of Resources: Memory Space, CPU Cycles, files, locks, Sockets, I/o devices etc.,

⊙ How a Process/thread utilize a Resource?

1) Request :- Request 'R', if 'R' is free lock it, else wait till it is available.
2) Use
3) Release :- Release Resource instance & make it available for other Process.



⊙ Deadlock Necessary Condition :-

4 Condition Should hold Simultaneously

a) **Mutual Exclusion**:- only 1 P at a time can use resource, if another Process request that resource → it should wait.

b) **Hold and wait**:- A Process must be holding at least 1 Resource & waiting to acquire additional resources that are currently being held by other Process.

c) **No - Preemption**:- Resource must be voluntarily released by Process after completion of Execution. (No Resource Preemption)!

d) **Circular wait**:- A Set {P0, P1, ... Pn} of waiting Process must Exist such that P0 is waiting for a resource held by P1, P1 is waiting for a resource held by P2 & so on...

→ **Methods for handling Deadlocks**:

a) use a Protocol to Prevent / avoid deadlocks, ensuring that the System will never enter a deadlock state

b) Allow the System to enter a deadlock state, detect it, & recover.

c) Ignore the Problem altogether & Pretend that deadlocks never occur in System (Ostrich algorithm) aka "**Deadlock ignorance**".

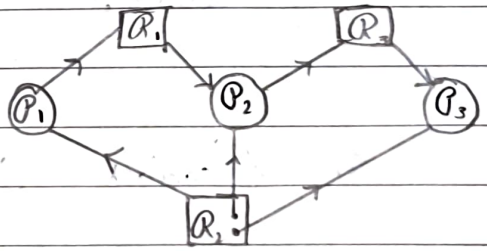
○ **Resource allocation Graph**:

1) Vertex: i) Process Vertex (P)
ii) Resource Vertex [R]

2) edges: i) Assign R1 (has) → P1
ii) Request P1 (Req) → R1

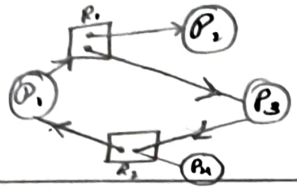
3) Multiple instances: R1 (4 instances) → P1 (4 Core CPU)

④ If Cycle is Present (RAG). (maybe)
↳ Deadlock is Present



⊙ Cycle's Present
⇒ D.L Present

(2)



⇒ After $t = 0$; P_4 & R_2 is released

⇒ R_2 is allotted to P_3 Now there No D.L is Seen

__/__/____

* In RAG; if Cycle is Present → D.L Maybe Present.
* In RAG; if cycle is not Present → D.L is not Present.

→ Deadlock Prevention & Deadlock avoidance Scheme:

(Not allowing Syster to enter D.L) (Make atleast 1 Condition not applied simultaneously)

i) Mutual Exclusion: (MX)

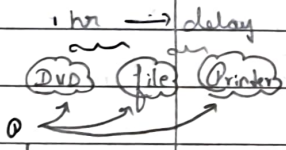
<C.S>← i) use locks (mutual Exclusion) only for non-Sharable Resources

ii) Sharable Resources like Read-only files can be accessed by multiple Process / threads (not using mutex).

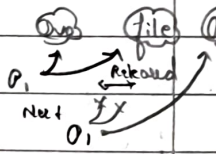
iii) However, we can't Prevent D.L by denying MX Condition, because Some resources are intrinsically non-Sharable.

ii) Hold & wait:

i) To ensure Hfw Condition never occurs in the System, we must guarantee that, whenever a Process request a resource, it doesn't hold any other Resource.



ii) Protocol (A) can be, each Process has to request and be allocated all its resources before its Execution.



iii) Protocol (B) can be, allow a Process to request resources only when it has none. It can request any additional resources after it must have released all the resources that it is Currently allocated

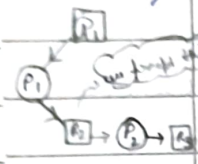
iii) No-Preemption:

i) If a Process is holding Some Resources & Request another resource that cannot be immediately allocated to it, then all Resources that Process is Currently holding are Preempted

__/__/__

* The Process will restart only when it can regain its old resources, as well as the new one that it is requesting (Else Lock may occur).

*) If a Process requests some resources, we first check whether they are available. If yes, we allocate them. If not, we check whether they are allocated to some other Process that is waiting for additional resources. If so, Preempt the desired resource from waiting Process and allocate them to the requesting Process



iv) Circular wait:

* To ensure that this condition never holds is to impose a proper ordering of resource allocation.

* P_1 & P_2 both require R_1 & R_2, locking on these resources should be like, both try to lock R_1 then R_2. By this way which ever Process 1st locks R_1 will get R_2.

⟹ **Deadlock Avoidance:**

→ Idea is, the kernel be given in advance info concerning which resources will use in its lifetime.

* By this, System can decide for each request whether the Process should wait (or) not.

* To decide whether the current request can be satisfied (or) delayed, the System must consider the
i) resources currently available.
ii) Resources currently allocated to each Process in the System
iii) Future Requests and Release of each Process.

* Current State: 1) No of Processes 2) Max need of R. of each Process
3) Currently allocated amount of R to each Process
4) Max amount of each R.

a) Schedule Process and its Resources allocation in such a way that the DL never occur.

b) Safe State: A State is safe if the System can allocate resources to each Process (up to its max.) in some order and still avoid DL.

* $P_1 \rightarrow P_3 \rightarrow P_2 \rightarrow P_4$,

* A System is in Safe State only if there exists Safe Sequence.

c) In an unsafe state, the O.S cannot Prevent Process from requesting resources in such a way that any deadlock occurs. It is not necessary that all unsafe States are deadlocks; an unsafe State may lead to a deadlock.

d) The main key of the deadlock avoidance method is whenever the request is made for Resources then the request must be only approved in the case if the resulting State is a Safe State.

e) In case, if the System is unable to fulfill the request of all Processes, then the State of the System is called unsafe.

f) Scheduling algorithum using which DL can be avoided by finding Safe State → 'Banker Algorithum'.

* Banker Algorithm:

* when a Process requests a set of resources, the System must determine whether allocating these resources will leave the System in a Safe State. If yes, resources may be allocated to the Process. If not, process must wait till other Processes release enough Resources.

P	Allocated R.			Max. need R			available (total - avai)			Remaining need (max R - allocated)			
	X	B	C	X	B	C	X	B	C	X	B	C	
P_1	0	1	0	7	5	3	10-7 = 3	5-2 = 3	7-5 = 2	7	4	3	✗ ④
P_2	2	0	0	3	2	2	+2 = 5	0 = 3	0 = 2	1	2	2	✓ ①
P_3	3	0	2	9	0	2	+2 = 7	+1 = 4	+1 = 3	6	0	0	✗ ⑤
P_4	2	1	1	4	2	2	+0 = 7	+0 = 4	+2 = 5	2	1	1	✗ ②
P_5	0	0	2	5	3	3	+0 = 7	+1 = 5	+0 = 5	5	3	1	✗ ③
							+2 = 10	+0 = 5	+0 = 7				

→ (Total avail) : 7 2 5 Let : Total : X=10, B=5 & C=7

* P_1 Cannot be Scheduled Since $7^{(need)} > 3^{(avai)}$
* P_2 is Scheduled Since (1<3) (2<3) & (2<=2)
* P_2 Completion later add it allocated R to available R.
* P_3 Cannot be * P_4 is Scheduled *
∴ $P_2 \rightarrow P_4 \rightarrow P_5 \rightarrow P_1 \rightarrow P_3$ ∴ It is a Safe State & dL Can be Prevented.

Some unsafe means :- If P_3 remaining is: 8, 0, 0 and at last we had 7, 1, 1 ⇒ It cannot be Scheduled thus leading to unsafe State

⊙ **Deadlock Detection:**

Systems haven't implemented dL-Prevention & a dL avoidance technique, then they may employ DL detection then, Recovery technique.

a) Single Instance of each Resource type (wait for graph method)

* A DL Exists in the System if & only if there is a Cycle in the 'wait-for graph'. In order to detect the DL, the System needs to maintain the wait-for graph and Periodically System invokes an algo. that Searches for the Cycle in the wait-for graph.

∴ DL is present

__/__/__

b) Multiple Instances for each resource type:
* Banker Algorithm. * Saf Sequence → also DL

*) Recovery from Deadlock:
a) Process Termination:
i) abort of all DL Process (forceful killing).
ii) Abort one process at a time until DL Cycle is eliminated.
b) Resource Preemption:
i) To eliminate DL, we Successively Preempt Some resources from Process and give these resources to other Process until DL cycle is broken.

*) <System> → <Algorithm> → DL? —no→ Run algo again after Some time
↓ Yes
<Recovery>