Deadlock Avoidance:

Deadlock prevention is used to prevent any of the conditions that cause deadlock. However, it results in degraded system efficiency and low device utilization. Therefore, this method cannot always be implemented.

Another mechanism for avoiding deadlock is deadlock avoidance, which checks in advance, the condition that may give rise to a deadlock. It indicates the state of a system, such that if the request of a process for a resource gives rise to a deadlock condition, it is denied, and must wait. In this way, the deadlock is avoided.

If the state of the system is such that it does not lead to a deadlock, then it is known as safe state. The converse, that is deadlock state is known as an unsafe state. The algo is run to check whether the requested resource changes the state of the system.

The deadlock avoidance algorithm, in the form of safe state protocol, must be run dynamically, whenever allocating a resource to a process.

In this way, deadlock avoidance approach is better than deadlock prevention, because it does not constrain the resources or processes, and there is no system performance degradation or device underutilization.

Deadlock Avoidance for Single Instance of Resource

To avoid a deadlock in a system, where every resource type has a single instance of resource, the RAG can be used again, but along with a new edge, known as claim edge. The claim edge is the same as request edge drawn from a process to a resource instance, but this does not mean that the request has been incorporated in the system. It is drawn in dotted lines. The RAG can be used in such a way that when a process requests for a resource, a corresponding claim edge is drawn, & the graph is checked before converting it to a request edge.

That is, a process request will not be entertained until the cycle check has been done. After the cycle check, if it is confirmed that there will be no circular wait, the claim edge is converted to a request edge. Otherwise, it will be rejected. In this way, the deadlock is avoided.

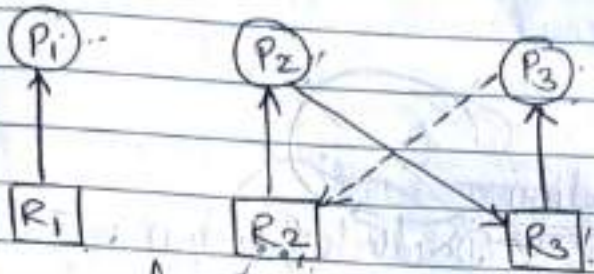

fig: RAG with claim edge

→ Dijkstra's Banker's Algorithm for Deadlock Avoidance in Multiple Instances of Resources

The RAG-based cycle check cannot be applied when there are multiple instances of resources, because in this case, it is not for certain that deadlock will occur. Therefore, an algorithm is designed to check the safe state, whenever a resource is requested. Dijkstra designed an algorithm, known as the Banker's algorithm i.e some data structures are maintained, such that whenever a resource is requested, it can be checked whether the request maintains the safe state of the system. If it does, the request can be granted. Otherwise, the process must wait until there are sufficient resources available.

The Banker's algorithm has two parts. The first part is Safety Test algorithm that checks the current state of the system for its safe state. The second part is resource request-handling algorithm that verifies whether the requested resources, when allocated to the process, affect the safe state. If it does, the request is denied. In this way, Banker's algorithm avoids the deadlock.

Q. Consider the following system snapshot using data structures in the Banker's algorithm.

Process	Max				Allocation				Available			
	A	B	C	D	A	B	C	D	A	B	C	D
P0	6	0	1	2	4	0	0	1	3	2	1	1
P1	1	7	5	0	1	1	0	0				
P2	2	3	5	6	1	2	5	4				
P3	1	6	5	3	0	6	3	3				
P4	1	6	5	6	0	2	1	2				

Using Banker's algo, answer the foll" questions:

i) How many resources of type A, B, C and D are there?

ii) What are the contents of need matrix?

iii) Is the system is in safe state? find the state sequence if it

Soln:

Resource = Allocation + Available

= 6 11 9 10 + 3 2 1 1

= 9 13 10 11

A B C D

Need

	A	B	C	D
P0	2	0	1	1
P1	0	6	5	0
P2	1	1	0	2
P3	1	0	2	0
P4	1	4	4	4

Sequence:

P0 <7 12 1 2>

P2 <8 4 6 6>

P3 <8 10 9 9>

P4 <8 12 10 11>

P1 <9 13 10 11>

* Banker's Algorithm:

n - processes

m - resources

Available$_m$: Available [i] = K

Max$_{n \times m}$: Max [i, j] = K

Allocation$_{n \times m}$: Allocation [i, j] = K

Need$_{n \times m}$: Need [i, j] = K

Algo: $P_i \rightarrow$ Request$_i$

1. If Request$_i \leq$ Need$_i$ then go to step 2 else error.
2. If Request$_i \leq$ Available then go to step 3 else wait.
3. Available = Available − Request$_i$ } System pretends
   Allocation$_i$ = Allocation + Request$_i$ } to have allocated
   Need$_i$ = Need$_i$ − Request$_i$ } the resources.
4. Check if this new state is safe i.e if a safe sequence exists.

Safety Algo:

1) Work = Available
   Finish = False.

2) Find an i such that
   Finish[i] = false and Need$_i$ ≤ Work
   if no such i, go to step 4.

3) Work = Work + Allocation
   Finish[i] = True
   go to step 2

4) If finish[i] = true for all i than the system is safe

:

A 10
B 5
C 7
} Resources

5 Processes
P_0 P_1 P_2 P_3 P_4

	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			

find Need :

Need = Max − Allocation
Work = Available + Allocation

Need

	A	B	C
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

Safe Sequence:

$<P_1, P_3, P_4, P_0, P_2>$

Available = Work = 3 3 2

P_1 : finish[1] = T
Work = 5 3 2

P_3: finish[3] = T
Work: 7 4 3

P_4: finish[4] = T
Work: 7 4 5

P_0: finish[0] = T
Work: 7 5 5

P_2: finish[2] = T
Work: 10 5 7

~~Allocation~~ Max

	R1	R2	R3
P_1	5	6	3
P_2	8	5	6
P_3	4	9	2
P_4	7	4	3
P_5	4	3	3
Total:	15	8	8

~~Max~~ Allocation

	R1	R2	R3
	2	1	0
	3	2	3
	3	0	2
	3	2	0
	1	0	1
	12	5	6
Remain	3	3	2

Need

	R1	R2	R3
P_1	3	5	3
P_2	5	3	3
P_3	1	9	0
P_4	4	2	3
P_5	3	3	2

P_5 : finish : 4 3 3
P_4 : finish : 7 5 3
P_1 : finish : 9 6 3
P_2 : finish : 12 8 6
P_3 : finish : 15 8 8

Safeseq: P5, P4, P1, P2, P3

Req = [2 0 2] for P4

Request ≤ Need → Yes

Request ≤ Available → Yes

Available = ~~3 3 2~~ Available − Request

= 3 3 2 − 2 0 2 = 1 3 0

Allocation = Allocation + Request

= 3 2 0 + 2 0 2 = 5 2 2

Need = Need − Request

= 4 2 3 − 2 0 2 = 2 2 1

→ ——— X

* Deadlock Detection

R1	R2	R3	} Total Res.
5	6	4	

	Allocation			Need		
	R1	R2	R3	R1	R2	R3
P1	1	0	2	1	0	0
P2	1	1	0	4	0	2
P3	1	1	0	0	0	2
P4	0	2	1	2	1	0
P5	1	2	0	3	1	4
	4	6	3			
Avail	1	0	1			

Safe seq: P1

P1 Avail < 2 0 3 >

P3 Avail < 3 1 3 >

P4 Avail < 3 3 4 >

P5 Avail < 4 5 4 >

P2 Avail < 5 6 4 >

New Req = P4 < 2 2 0 >

P1 Avail < 2 0 3 >

P3 Avail < 3 1 3 >

Deadlock