a- Banker’s Algorithm

This Alg. Could be used in a banking system to ensure that the bank never allocates its available cash in such a way that it can no longer satisfy the needs of all its customers. When a new a process enters the system it must declare the maximum number of instances of each resource type that it may need.

- The maximum must be ≤ total number of resources in the system.

-When a user requests a set of resources must be leave the system in a safe state if the resources are allocated otherwise the process must wait until some other process releases enough resources.

Several data structures must be maintained to implement banker’s algorithm
Let n be the number of processes in the system and m be the number of resource types . We need the following data structures:

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

available . If available [j] = k these are k instances of resource type R_j

- 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
- Allocation : An $n_x m$ the resources currently allocated to each process . If allocation [i,j] = k then process p_i is currently allocated process p_i is currently allocated 1 instances of resources of resource type $R_{j.}$
- Need : An $n_x m$ 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]

1- If request i ≤ Need i go to step 2 otherwise raise an error since the process has exceeded its maximum claim.

2- If request i ≤ Available go to step 3 . Otherwise p_i must wait since the resources are not available .

3- The system pretends to have allocated the requested resources to process p_i by modifying the state as follows:

Available := Available – Requesti ,
Allocationi := Allocationi + Requesti ,

Needi:= Needi – Requesti,
If the resulting resource allocation state is safe the transaction is completed and process p_i is allocated its resources. If the new state is unsafe the p_i must wait for request i and the old resource allocation state is restored.

c- Safety Algorithm

The algorithm for finding out whether or out a system is in a safe state can be described as follows:

1- Let <u>work</u> and <u>finish</u> be vectors of length m and n respectively.
   Initialize work := Available and Finish [i] := False for I = 1,2, …, n
2- Find an i such that both
- Finish [i] = false
- Need i ≤ work

If no such i exits, go to step 4

3- Work := work +allocation I Finish[i] := true go to step2
4- If Finish [i] = true for all I then the system is in a safe state

This algorithm may require an order of $m_x n^2$ operations to decide whether a state is safe.

<u>Example:</u>

Consider a system with five processes $\{p_0,p_1,p_2,\ldots\}$ and three resource types {A,B,C} . Resource type A has 10 instances. Resource type B has 5 instances, and resource type c has 7 instance. 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	
0 1 0	7 5 3	3 3 2	P_0
2 0 0	3 2 2		P_1
3 0 2	9 0 2		P_2
2 1 1	2 2 2		P_3
0 0 2	4 3 3		P_4

The content of the matrix Need is defined to be max-Allocation and is:

	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

The system is in the safe state if the processes executed in the sequence (p_1, p_3 , p_4 ,p_2 , p_0).
Suppose now that process p_i requests one additional instance of resource type A and two instance of resources type C so request 1= (1 ,0 , 2)
To decide whether this request can be immediately granted we first check that
Request 1 ≤ Available (that is , (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	Max	Available	
A B C	A B C	A B C	
0 1 0	7 4 3	3 3 2	**P_0**
3 0 2	0 2 0		**P_1**
3 0 2	6 0 0		**P_2**
2 1 1	0 1 1		**P_3**
0 0 2	4 3 1		**P_4**

By execute the safety Alg. We find the sequence (p_1 , p_3 , p_4 , p_0 , p_2) satisfies our safety requirements . Hence we can immediately grant the request of process p_1

If p_4 request for (3 , 3 , 0) . The request can not granted since the resources are not available . Request 1> Available . If p_0 request (0 , 2 , 0) can not granted even though the resources are available since the resulting state is unsafe.

Deadlock Detection

If a system does not employ some protocol that ensures that no deadlock will never occur. Then a detection and recovery scheme must be implemented . The system can use an algorithm to examines the state of the system periodically to determine whether has occurred . If so the system must recover from the deadlock by providing :

a- Maintain information about the current allocation of resources to processes and outstanding request.

b- Provide an Alg. That use the above information to determine whether the system has entered the deadlock state.

The detection Alg. Employs several time – varying data structures that are very similar to those used in the Banker's Algorithm :

- **Available**
- **Allocation**
- **Request** . An $n_x m$ matrix indicating the current request of each process.
  If Request [i,j] = k then p_i is requesting k more instances of resource type r_j .
  The detection Alg. Simply investigates every possible allocation sequence for the processes that remain to be completed.

The detection Alg . as follows:

1- Let work and finish be vectors of length m and n respectively . Initialize
Work := Available , for i = 1 ,2 ,3, … , n. If allocation $\neq 0$ the Finish [i] := false.
Otherwise , Finish[i] := false.

2- Find an index i such that :

- Finish [i] = false , and
- Request i $\leq$ work .

If no such I exits go to step 4.

3- Work := work +Allocation i
Finish [i] := true
Go to step2

4- If Finish [i] = false , for some i , $1 \leq i \leq n$ then the system is in a deadlock state. More over , if Finish [i] = false then process p_i is deadlocked.

Example

Consider a system with five processes {p_0 , p_1 ,… p_4} and three resources types { A=7 instance , B =2 , C=6 instance} suppose that at time T_0 we the following resource allocation state.

Allocation	Max	Available	
A B C	A B C	A B C	
0 1 0	0 0 0	0 0 0	**P_0**
2 0 0	2 0 2		**P_1**
3 0 3	0 0 0		**P_2**
2 1 1	1 0 0		**P_3**
0 0 2	0 0 2		**P_4**

If we execute the detection Alg. We find the system is not in a deadlock state and the sequence < p_0 , p_2 , p_3 , p_1 ,p_4 > will result in finish [i] = true for all i .

Suppose now that process p2 makes one additional request for an instance of type C. the Request matrix is modified as follows :

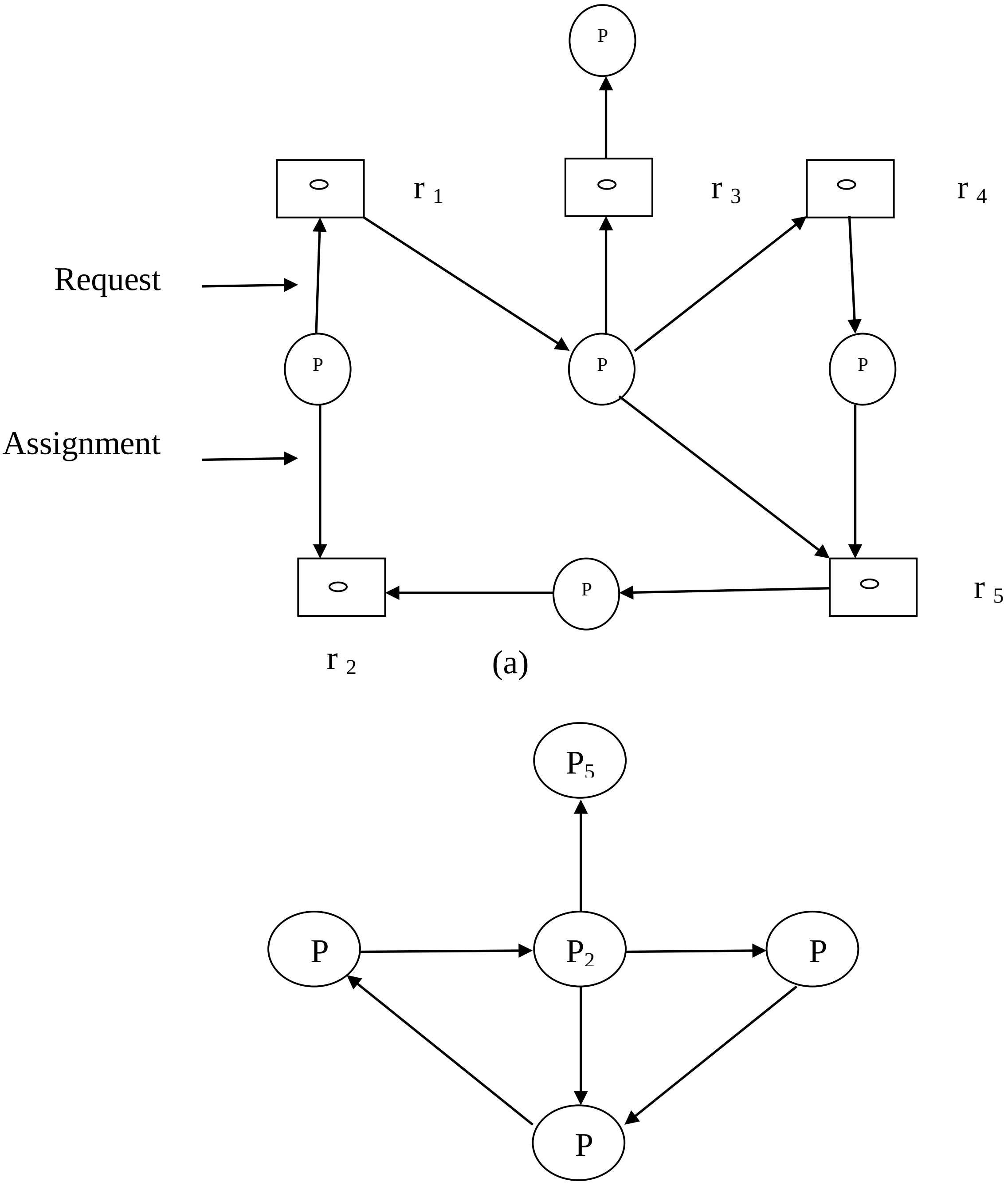
	Need
	A B C
P_0	0 0 0
P_1	2 0 2
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

We claim that the system is now deadlocked . Although we can reclaim the resources held by process p_0 the number of available resources is not sufficient to fut fill the requests of the other processes . Thus a deadlock exist , consisting of processes < p_1 , p_2 , p_3 and p_4 >.

- Single Instance of each resource type

  The detection Alg. is of order $m_x n^2$. If all resources have only a single instance we can define a faster Alg. we will use a variant of the resource allocation graph called a wait – for graph . This graph is obtained from the resource allocation graph by removing the nodes of type resource and collapsing the appropriate edges. Where the edge from p_i is waiting for process p_j to release a resource that it needs.

  An edge (p_i ,p_j) exists in a wait – for graph if and only if the resource RAG contains two edges (p_i , r_q) and (r_q , p_i) for some resource , see fig bellow



(a)

(b)

Fig: RAG (a) and its wait – for graph (b)

A deadlock exists in the system if and only if the wait – for graph contains a cycle.

- **Recovery from deadlock**

  When a detection Alg. determines that a deadlock exists the system must recover from the deadlock .

  There are two options for breaking a deadlock

  a- **Process termination** by killing a process , two methods:
- Kill all deadlocked processes.
- Kill one process at a time until the deadlock cycle is eliminated.

b- **Resource preemption**, to eliminate deadlocks using resource preemption we can preempt some resources from processes and give them to other processes until the deadlock cycle is broken .

If preemption is required in order to deal with deadlocks then three issues need to be addressed:

- Selecting a victim: which process and which resources .
- Rollback : if we preempt a resource from a process what should be done with that process?

- Starvation : How do we ensure that Starvation will not occur?
  That is how can we guarantee that resources will not always be preempted from the some process?