BANKER'S ALGORITHM

- The resource-allocation-graph algorithm is not applicable to a system with multiple instances of each resource type.
- Banker's algorithm is able to deal with multiple instances of different resource types.
- The name was chosen because the algorithm could be used in a banking system to ensure that the bank never allocated its available cash in such a way that it could no longer satisfy the needs of all its customers.
- 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 of 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.
- Several **data structures** must be maintained to implement the banker's algorithm.
- We need the following data structures, where *n* is the number of processes in the system and *m* is the number of resource types:

1. **Available**. A vector of length *m* indicates the number of available resources of each type. If ***Available***[*j*] equals *k*, then *k* instances of resource type *Rj* are available.
2. **Max**. An $n \times m$ matrix defines the maximum demand of each process. If ***Max***[*i*][*j*] equals *k*, then process *Pi* may request at most *k* instances of resource type *Rj* .
3. **Allocation**. An $n \times m$ matrix defines the number of resources of each type currently allocated to each process. If ***Allocation***[*i*][*j*] equals *k*, then process *Pi* is currently allocated *k* instances of resource type *Rj* .
4. **Need**. An $n \times m$ matrix indicates the remaining resource need of each process. If ***Need***[*i*][*j*] equals *k*, then process *Pi* may need *k* more instances of resource type *Rj* to complete its task.

- Note that ***Need***[*i*][*j*] = ***Max***[*i*][*j*] − ***Allocation***[*i*][*j*].
- Banker's Algorithm uses 2 sub-algorithms
    1. Safety Algorithm
    2. Resource-Request Algorithm

Safety Algorithm

1. Let ***Work*** and ***Finish*** be vectors of length *m* and *n*, respectively. Initialize ***Work*** = ***Available*** and ***Finish***[*i*] = ***false*** for $i = 0, 1, ..., n - 1$.

2. Find an index *i* such that both

a. ***Finish***[*i*] == ***false***

b. ***Need****i* ≤***Work***

If no such *i* exists, go to step 4.

3. ***Work*** =***Work*** + ***Allocation***i
   Finish[i] = ***true***
   Go to step 2.
4. If ***Finish***[i] == ***true*** for all i, then the system is in a safe state.

- This algorithm may require an order of $m \times n^2$ operations to determine whether a state is safe.

Resource-Request Algorithm

- This algorithm is used for determining whether requests can be safely granted.
- Let ***Request****i* be the request vector for process *Pi*.
- When a request for resources is made by process *Pi* , the following actions are taken:

1. If ***Request****i* ≤ ***Need****i* , go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If ***Request****i* ≤ ***Available,*** go to step 3. Otherwise, *Pi* must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to process *Pi* by modifying the state as follows:

   Available = ***Available*** – ***Request****i* ;

   Allocation*i* = ***Allocation****i* + ***Request****i* ;

   Need*i* = ***Need****i* – ***Request****i* ;

- If the resulting resource-allocation state is safe, the transaction is completed, and process *Pi* is allocated its resources.
- However, if the new state is unsafe, then *Pi* must wait for ***Requesti* ,** and the old resource-allocation state is restored.

<u>Problem</u>

- Consider a system with five processes P0 through P4 and three resource types A, B, and C. Resource type A has ten instances, resource type B has five instances, and resource type C has seven instances. Suppose that, at time T0, 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	

(a) Find whether the system is safe. If yes, give the safe sequence

(b) Suppose that process P1 requests one additional instance of resource type A and two instances of resource type C. Whether it can be immediately granted?

(c) After granting (b), if P4 request for (3,3,0), whether it can be granted?

(d) Can a request for (0,2,0) by P0 be granted?

Solution

Need* = *Max* – *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

Solution (a)

Run Safety Algorithm

Work = Available = (3,3,2)

Consider P0
Need0 = (7,4,3)
Need0 (7,4,3) not less than Work (3,3,2)

Consider P1
Need1= (1,2,2) is less than Work (3,3,2) //each element is less. So we can complete P1
Work = Work + Allocation1
ie Work = (3,3,2) + (2,0,0) = (5,3,2)
Set Finish[1] = true **//P1 finished**

Consider P2
Need2 (6,0,0) not less than Work (5,3,2)

Consider P3
Need3 (0,1,1) is less than Work (5,3,2)
So we can complete P3
Work = Work + Allocation3
Work = (5,3,2) + (2,1,1) = (7,4,3)
Set Finish[3] = true **//P3 finished**

Consider P4
Need4 (4,3,1) is less than Work (7,4,3)
So we can complete P4
Work = Work + Allocation4
Work = (7,4,3) + (0,0,2) = (7,4,5)
Set Finish[4] = true **//P4 finished**

Consider P0
Need0 (7,4,3) is less than Work (7,4,5)
So we can complete P0
Work = Work + Allocation0
Work = (7,4,5) + (0,1,0) = (7,5,5)
Set Finish[0] = true **//P0 finished**

Consider P2
Need2 (6,0,0) is less than Work (7,5,5)
So we can complete P2
Work = Work + Allocation2
Work = (7,5,5) + (3,0,2) = (10,5,7)
Set Finish[2] = true **//P2 finished**

We could successfully complete all the processes.
So the system is safe and the safe sequence is the order of completion <P1, P3, P4, P0, P2>

At last the Work becomes the initial counts of resources A, B and C as (10,5,7)

Solution (b)

Suppose that process P1 requests one additional instance of resource type A and two instances of resource type C.

Form Request vector Request1 = (1,0,2) and run Resource – Request Algorithm

Request1 (1,0,2) is less than Need1 (1,2,2)
Request1 (1,0,2) is less than Available (3,3,2)
Pretend to have allocated
Available = Available – Request1 ;
// Available = (3,3,2) – (1,0,2) = (2,3,0)
Allocation1 = Allocation1 + Request1 ;
// Allocation1 = (2,0,0) + (1,0,2) = (3,0,2)
Need1 = Need1 – Request1 ;
// Need1 = (1,2,2) - (1,0,2) = (0,2,0)

New state is shown below:

	Allocation	*Need*	*Available*
	A B C	*A B C*	*A B C*
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

While running safety algorithm *(home work)*, we get a safe sequence <P1, P3, P4, P0, P2>.So the system is safe.
Hence the request can be immediately granted

Solution (c)

If P4 request for (3,3,0), whether it can be granted?
Request4 (3,3,0) is less than Need4 (4,3,1)
But Request4 (3,3,0) is not less than Available (2,3,0)
So as per step2 of Request algorithm, **the request cannot be immediately granted. P4 should wait.**

Solution (d)

Can a request for (0,2,0) by P0 be granted?

Request0 (0,2,0) is less than Need0 (7,4,3)
Request0 (0,2,0) is less than Available (2,3,0)
Pretend to have allocated
Available = Available – Request0 ;
// Available = (2,3,0) – (0,2,0) = (2,1,0)
Allocation0 = Allocation0 + Request0 ;
// Allocation0 = (0,1,0) + (0,2,0) = (0,3,0)
Need0 = Need0 – Request0 ;
// Need0 = (7,4,3) - (0,2,0) = (7,2,3)

New state is shown below:

	Allocation	Need	Available
	A B C	A B C	A B C
P_0	0 3 0	7 2 3	2 1 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

Run safety algorithm. No process can complete. So no safe sequences exist. **So system is unsafe. Hence request cannot be granted.**

DEADLOCK DETECTION ALGORITHM

- **Similar to Bankers algorithm**
- In Bankers algorithm (deadlock avoidance), future information (***Max*** & ***Need***) is considered. But during deadlock detection, future information is not needed. Only the present request is considered.
- All data structures in detection algorithms are same as that of Bankers algorithm except in the case of ***Max*** & ***Need***.
- Here we use a matrix ***Request*** instead of ***Need***. Max is avoided
- **Deadlock detection Algorithm**:
  [Write Bankers algorithm by renaming Need as Request]

Problem

Consider a system with five processes P0 through P4 and three resource types A, B, and C. Resource type A has seven instances, resource type B has two instances, and resource type C has six instances. Suppose that, at time T0, we have the following resource-allocation state:

	Allocation	Request
	A B C	A B C
P_0	0 1 0	0 0 0
P_1	2 0 0	2 0 2
P_2	3 0 3	0 0 0
P_3	2 1 1	1 0 0
P_4	0 0 2	0 0 2

(a) Check whether the system is in deadlock or not

(b) Suppose that process P2 makes one additional request for an instance of type C. Check whether it leads to deadlock? If yes, find the processes involved in deadlock.

Solution

First we have to find out Available
Total allocated number of resource type A is 0+2+3+2+0=7
Total allocated number of resource type B is 1+0+0+1+0=2
Total allocated number of resource type C is 0+0+3+1+2=6
So remaining is (0,0,0) which is our Available

Solution (a)

Run safety algorithm and we can find a safe sequence <P0, P2, P3, P4, P1> **So system is safe, hence not in deadlock**

Solution (b)

Additional Request2 = (0,0,1). It may be added with current request of P2. Request matrix is updated as below

	Request
	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

Run safety algorithm. We didn’t get a safe sequence. Only P0 is completed.

Hence the system is in deadlock situation and the processes involved in deadlock are P1, P2, P3 and P4