**AIM:** Banker’s algorithm for deadlock avoidance and detection.

**THEORY:**

DEADLOCK: A deadlock involving a set of processes D is a situation in which

1. Every process pi  in D is blocked on some event ei.
2. Event ei can only be caused by some process (es) in D.

A long delay in the occurrence of an awaited event , i.e. starvation of some kind ,cannot contribute to a deadlock. . The important requirement is that processes capable of causing an event must themselves belong to D. This makes it impossible for the event to happen.

Deadlock affects the progress of processes by causing indefinite delays in resource allocation.

Such delays have serious consequences for the response times of processes, idling and wastage

of resources allocated to processes, and the performance of the system.

HANDLING DEADLOCKS:

Two fundamental approaches used for handling deadlocks are:

1. Detection and resolution of deadlocks.
2. Avoidance of deadlocks.

In the former approach, the OS detects deadlock situations as and when they arise. It then

performs some actions aimed at ensuring progress for some of the deadlocked processes.

These actions constitute deadlock resolution. The latter approach focuses on on avoiding the

occurrence of deadlocks. This approach involves checking each resource request to ensure

that granting does not lead to deadlock. The detection and resolution approach does not perform

any such checks.

**Banker's algorithm**

The **Banker's algorithm** is a resource allocation & deadlock avoidance algorithm developed by Edsger Dijkstra that tests for safety by simulating the allocation of pre-determined maximum possible amounts of all resources, and then makes a "safe-state" check to test for possible deadlock conditions for all other pending activities, before deciding whether allocation should be allowed to continue.

The Banker’s algorithm uses an avoidance policy. Hence no validity constraint is specified,

and processes are free to make any resource requests. The system uses two tests – *feasibility*

*test* and  *safety test.* – to govern its resource allocation actions when process pj makes a req-

uest Reqj,k for units of resource class k.

**Feasible request :** A feasible request if, after granting the request, the allocated resources of class k do not exceed the total number of resource units of class k in the system.

**Safe request :** A feasible request Reqj,k is a safe request at time t if, after granting

The request, there exists at least one sequence of allocations and releases by which all processes in the system can complete.

The system grants only those requests which are both feasible and safe. All other request are kept pending till they become both feasible and safe.

The OS uses a common sense admission criterion while initiating a process pj , namely that the maximum no. of units of resource class k required by pj at any time during its execution must not exceed the total no. of units of resource class k in the system . The OS may initiate any no. processes satisfying the admission criteria. Thus the total resource requirements of all admitted processes may exceed the no. of resource units of resource class k. the algorithm is called bankers algorithm presumably because bankers follow an analogous procedure while granting loans.

#### Example

Assuming that the system distinguishes between four types of resources, (A, B, C and D), the following is an example of how those resources could be distributed. *Note that this example shows the system at an instant before a new request for resources arrives. Also, the types and number of resources are abstracted. Real systems, for example, would deal with much larger quantities of each resource.*

Available system resources:

A B C D

3 1 1 2

Processes (currently allocated resources):

A B C D

P1 1 2 2 1

P2 1 0 3 3

P3 1 1 1 0

Processes (maximum resources):

A B C D

P1 3 3 2 2

P2 1 2 3 4

P3 1 1 5 0

### Safe and Unsafe States

A state (as in the above example) is considered safe if it is possible for all processes to finish executing (terminate). Since the system cannot know when a process will terminate, or how many resources it will have requested by then. The system assumes that all processes will eventually attempt to acquire their stated maximum resources and terminate soon afterward. This is a reasonable assumption in most cases since the system is not particularly concerned with how long each process runs (at least not from a deadlock avoidance perspective). Also, if a process terminates without acquiring its maximum resources, it only makes it easier on the system.

Given that assumption, the algorithm determines if a state is safe by trying to find a hypothetical set of requests by the processes that would allow each to acquire its maximum resources and then terminate (returning its resources to the system). Any state where no such set exists is an unsafe state.

#### Example

We can show that the state given in the previous example is a safe state by showing that it is possible for each process to acquire its maximum resources and then terminate.

1. P1 acquires 2 A, 1 B and 1 D more resources, achieving its maximum

The system now still has 1 A, no B, 1 C and 1 D resource available

2. P1 terminates, returning 3 A, 3 B, 2 C and 2 D resources to the system

The system now has 4 A, 3 B, 3 C and 3 D resources available

3. P2 acquires 2 B and 1 D extra resources, then terminates, returning all its resources

The system now has 5 A, 3 B, 6 C and 6 D resources

4. P3 acquires 4 C resources and terminates

The system now has all resources: 6 A, 4 B, 7 C and 6 D

5. Because all processes were able to terminate, this state is safe

Note that these requests and acquisitions are *hypothetical*. The algorithm generates them to check the safety of the state, but no resources are actually given and no processes actually terminate. Also note that the order in which these requests are generated – if several can be fulfilled – doesn't matter, because all hypothetical requests let a process terminate, thereby increasing the system's free resources.

For an example of an unsafe state, look at what would happen if process 2 were holding 1 more unit of resource B at the beginning.

### Requests

When the system receives a request for resources, it runs the Banker's algorithm to determine if it is safe to grant the request. The algorithm is fairly straight forward once the distinction between safe and unsafe states is understood.

1. Can the request be granted?

if not, the request is impossible and must either be denied or put on a waiting list

2. Assume that the request is granted

3. Is the new state safe?

If so grant the request

If not, either deny the request or put it on a waiting list

Whether the system denies an impossible or unsafe request or makes it wait is an operating system specific decision.

#### Example

Continuing the previous examples, assume process 3 requests 2 units of resource C.

1. There is not enough of resource C available to grant the request
2. The request is denied

On the other hand, assume process 3 requests 1 unit of resource C.

1. There are enough resources to grant the request
2. Assume the request is granted

The new state of the system would be:

A B C D

Free 3 1 0 2

P1 1 2 2 1

P2 1 0 3 3

P3 1 1 2 0

Determine if this new state is safe

1. P1 can acquire 2 A, 1 B and 1 D resources and terminate
2. Then, P2 can acquire 2 B and 1 D resources and terminate
3. Finally, P3 can acquire 3 C resources and terminate
4. Therefore, this new state is safe

Since the new state is safe, grant the request.

Finally, assume that process 2 requests 1 unit of resource B.

1. There are enough resources
2. Assuming the request is granted, the new state would be:

A B C D

Free 3 0 1 2

P1 1 2 2 1

P2 1 1 3 3

P3 1 1 1 0

Is this state safe? Assuming P1, P2, and P3 request more of resource B and C.

1. P1 is unable to acquire enough B resources

2. P2 is unable to acquire enough B resources

3. P3 is unable to acquire enough C resources

No process can acquire enough resources to terminate, so this state is not safe

Since the state is unsafe, deny the request

*Note that in this example, no process was able to terminate. It is possible that some processes will be able to terminate, but not all of them. That would still be an unsafe state.*

**Algorithm**

Inputs

n - Number of processes;

r - Number of resource classes

Blocked - Set of processes

Running - Set of processes

Preq - processes making the new resource request;

New request - array [ 1….r ] of integers ;

**Data structures**

Max : Array [ 1…n,1 ….r] of integer;

Allocated \_resources : Array [ 1…n, 1…r] of integers;

Requested\_resources : Array[ 1…n, 1…r] of integers;

Total\_ alloc : Array [1…r] of integers;

Total\_exist : Array [1…r] of integers;

Active : Set of processes;

Simulated\_alloc : Array [1…r] of integers;

1. Active : = Running U Blocked;
2. For k = 1….r

Allocated\_resources[req,k] : = New\_request[k] ;

1. For k = 1….r /\*compute projected state\*/

Allocated\_resources[req,k] : = Alllocated\_request[req,k]+ new\_request[k];

Total\_alloc[k] : = Total\_alloc[k] + new\_request[k];

4. For k = 1….r /\*check if new state is feasible\*/

if requested\_resources[req,k]> Total\_exist[k] – Total\_alloc[k]

**then go to step 6;**

1. Simulated\_alloc : = Total\_alloc;

While  Pl Є active such that k

Total\_exist[k]- simulated alloc[k] >=max[l,k] – allocated\_resoureces[l,k]

* 1. Delete Pl from active ;
  2. For k = 1…r

Simulated\_alloc[k] : = Simulated\_alloc[k] - allocated\_resoureces[l,k];

6. If active is empty **then** /\*Projected state is safe\*/

For k = 1….r

Requested\_resources[req,k] : = 0;

Else /\*disallow projected grant and revert to current state\*/

For k= 1….r

Allocated\_resources[ req,k] : = Allocated\_resources[ req,k] - new\_request[k];

Total\_alloc[k] : = Total\_alloc[k] - new\_request[k];

The algorithm keeps a request pending if the projected state is infeasible (step 4). Else it simulates the grant of the new request (step 3) and determines its safety (step 5).

If the request is safe , its grant is confirmed, else it is nullified (step 6).

## Trade-offs

Like most algorithms, the Banker's algorithm involves some trade-offs. Specifically, it needs to know how much of each resource a process could possibly request. In most systems, this information is unavailable, making the Banker's algorithm useless.

## Flowchart:-

## Conclusion:-