Lab # 8

Implementation of Deadlock Avoidance Banker's Algorithm and study about deadlock recovery

Banker's Algorithm is a deadlock avoidance algorithm that checks for safe or unsafe state of a System after allocating resources to a process.

Why Banker's Algorithm is Named So?

The banker's algorithm is named so because it is used in the banking system to check whether a loan can be sanctioned to a person or not. Suppose there are n number of account holders in a bank and the total sum of their money is S. If a person applies for a loan then the bank first subtracts the loan amount from the total money that the bank has and if the remaining amount is greater than S then only the loan is sanctioned. It is done because if all the account holders come to withdraw their money then the bank can easily do it.
It also helps the OS to successfully share the resources between all the processes. It is called the banker's algorithm because bankers need a similar algorithm- they admit loans that collectively exceed the bank's funds and then release each borrower's loan in installments. The banker's algorithm uses the notation of a safe allocation state to ensure that granting a resource request cannot lead to a deadlock either immediately or in the future. In other words, the bank would never allocate its money in such a way that it can no longer satisfy the needs of all its customers. The bank would try to be in a safe state always.

When a new process enters into system, it must declare maximum no. of instances of each resource that it may need. After requesting operating system run banker's algorithm to check whether after allocating requested resources system goes into deadlock state or not. If yes then it will deny the request of resources made by process else it allocate resources to that process.

No. of requested resources (instances of each resource) may not exceed no. of available resources in operating system and when a process completes it must release all the requested and already allocated resources.

Implementation of Banker's Algorithm:

For implementing Banker's algorithm, we should have pre-knowledge about three things:

- How many instances of each resource a process could request. (Max)
- How many instances of each resource is already allocated to that process (Allocated
- How many instances of each resource is already available (Available).

We can calculate need of each process from above information: Need=Max - Allocated.
If need<=available then request will be accepted otherwise it is denied and it will check for next process in waiting queue.(See Life Cycle Of a Process).

Safe or Unsafe State:

A system is in Safe state if its all process finish its execution or if any process is unable to acquire its all requested resources then system will be in Unsafe state.

Example:

Considering a system with five processes P0 through P4 and three resources of type A, B, C. Resource type A has 10 instances, B has 5 instances and type C has 7 instances. Suppose at time t0 following snapshot of the system has been taken:

Process	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	

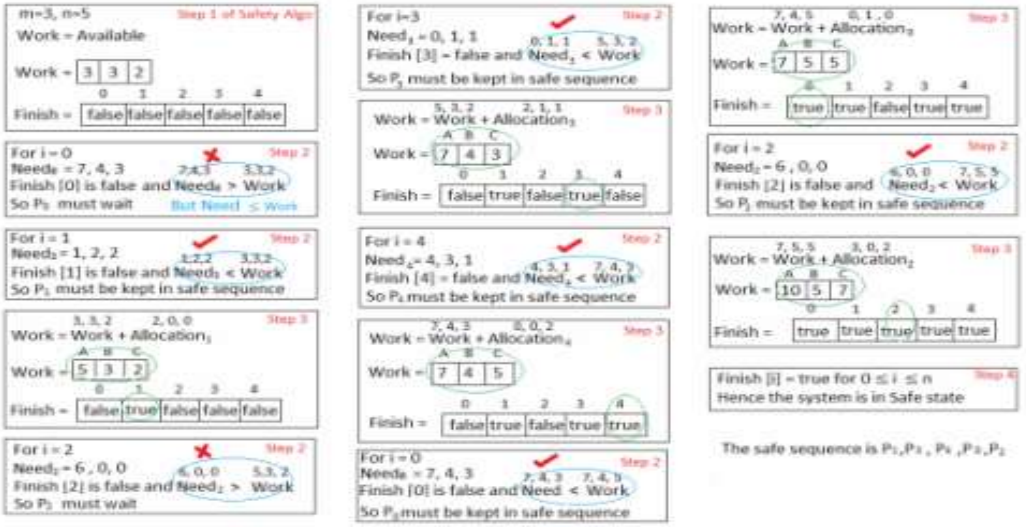
Q.1: What will be the content of the Need matrix?

Need [i, j] = Max [i, j] – Allocation [i, j]

So, the content of Need Matrix is:

Process	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

Q.2: Is the system in a safe state? If Yes, then what is the safe sequence?Applying the Safety algorithm on the given system,



Deadlock Detection And Recovery

Deadlock detection and recovery is the process of detecting and resolving deadlocks in an operating system. A deadlock occurs when two or more processes are blocked, waiting for each other to release the resources they need. This can lead to a system-wide stall, where no process can make progress.

There are two main approaches to deadlock detection and recovery:

Prevention: The operating system takes steps to prevent deadlocks from occurring by ensuring that the system is always in a safe state, where deadlocks cannot occur. This is achieved through resource allocation algorithms such as the Banker's Algorithm.

Detection and Recovery: If deadlocks do occur, the operating system must detect and resolve them. Deadlock detection algorithms, such as the Wait-For Graph, are used to identify deadlocks, and recovery algorithms, such as the Rollback and Abort algorithm, are used to resolve them. The recovery algorithm releases the resources held by one or more processes, allowing the system to continue to make progress.

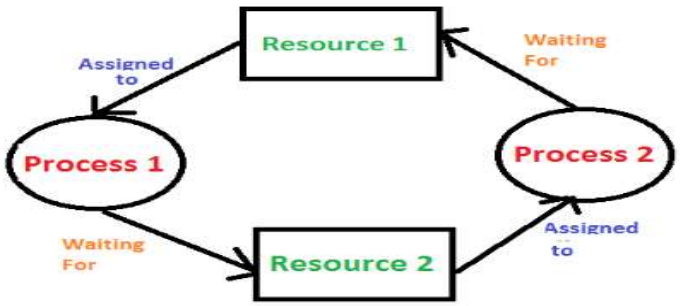
Difference Between Prevention and Detection/Recovery: Prevention aims to avoid deadlocks altogether by carefully managing resource allocation, while detection and recovery aim to identify and resolve deadlocks that have already occurred.

Deadlock detection and recovery is an important aspect of operating system design and management, as it affects the stability and performance of the system. The choice of deadlock detection and recovery approach depends on the specific requirements of the system and the trade-offs between performance, complexity, and risk tolerance. The operating system must balance these factors to ensure that deadlocks are effectively detected and resolved.

Deadlock Detection :

1. If resources have a single instance –

In this case for Deadlock detection, we can run an algorithm to check for the cycle in the Resource Allocation Graph. The presence of a cycle in the graph is a sufficient condition for deadlock.



In the above diagram, resource 1 and resource 2 have single instances. There is a cycle R1 → P1 → R2 → P2. So, Deadlock is Confirmed.

2. If there are multiple instances of resources –
Detection of the cycle is necessary but not a sufficient condition for deadlock detection, in this case, the system may or may not be in deadlock varies according to different situations.

3. Wait-For Graph Algorithm –

The Wait-For Graph Algorithm is a deadlock detection algorithm used to detect deadlocks in a system where resources can have multiple instances. The algorithm works by constructing a Wait-For Graph, which is a directed graph that represents the dependencies between processes and resources.

Deadlock Recovery:
A traditional operating system such as Windows doesn't deal with deadlock recovery as it is a time and space-consuming process. Real-time operating systems use Deadlock recovery.

Killing the process –
Killing all the processes involved in the deadlock. Killing process one by one. After killing each process check for deadlock again and keep repeating the process till the system recovers from deadlock. Killing all the processes one by one helps a system to break circular wait conditions.

Resource Preemption –
Resources are preempted from the processes involved in the deadlock, and preempted resources are allocated to other processes so that there is a possibility of recovering the system from the deadlock. In this case, the system goes into starvation.

Concurrency Control –
Concurrency control mechanisms are used to prevent data inconsistencies in systems with multiple concurrent processes. These mechanisms ensure that concurrent processes do not access the same data at the same time, which can lead to inconsistencies and errors. Deadlocks can occur in concurrent systems when two or more processes are blocked, waiting for each other to release the resources they need. This can result in a system-wide stall, where no process can make progress. Concurrency control mechanisms can help prevent deadlocks by managing access to shared resources and ensuring that concurrent processes do not interfere with each other.

ADVANTAGES OR DISADVANTAGES:
Advantages of Deadlock Detection and Recovery in Operating Systems:

Improved System Stability: Deadlocks can cause system-wide stalls, and detecting and resolving deadlocks can help to improve the stability of the system.

Better Resource Utilization: By detecting and resolving deadlocks, the operating system can ensure that resources are efficiently utilized and that the system remains responsive to user requests.

Better System Design: Deadlock detection and recovery algorithms can provide insight into the behavior of the system and the relationships between processes and resources, helping to inform and improve the design of the system.

Disadvantages of Deadlock Detection and Recovery in Operating Systems:

Performance Overhead: Deadlock detection and recovery algorithms can introduce a significant overhead in terms of performance, as the system must regularly check for deadlocks and take appropriate action to resolve them.

Complexity: Deadlock detection and recovery algorithms can be complex to implement, especially if they use advanced techniques such as the Resource Allocation Graph or Timestamping.

False Positives and Negatives: Deadlock detection algorithms are not perfect and may produce false positives or negatives, indicating the presence of deadlocks when they do not exist or failing to detect deadlocks that do exist.

Risk of Data Loss: In some cases, recovery algorithms may require rolling back the state of one or more processes, leading to data loss or corruption.

Overall, the choice of deadlock detection and recovery approach depends on the specific requirements of the system, the trade-offs between performance, complexity, and accuracy, and the risk tolerance of the system. The operating system must balance these factors to ensure that deadlocks are effectively detected and resolved.

C# Code for Banker's Algorithm:

```
// C# Program for Bankers Algorithm
using System;
using System.Collections.Generic;

class GFG
{
static int n = 5; // Number of processes
static int m = 3; // Number of resources
int [,]need = new int[n, m];
int [,]max;
int [,]alloc;
int []avail;
int []safeSequence = new int[n];

void initializeValues()
{
    // P0, P1, P2, P3, P4 are the Process
    // names here Allocation Matrix
    alloc = new int[,] {{ 0, 1, 0 }, //P0
                        { 2, 0, 0 }, //P1
                        { 3, 0, 2 }, //P2
                        { 2, 1, 1 }, //P3
                        { 0, 0, 2 }};//P4

    // MAX Matrix
    max = new int[,] {{ 7, 5, 3 }, //P0
                        { 3, 2, 2 }, //P1
                      { 9, 0, 2 }, //P2
                      { 2, 2, 2 }, //P3
                      { 4, 3, 3 }};//P4
```

```
    // Available Resources
    avail = new int[] { 3, 3, 2 };
}

void isSafe()
{
    int count = 0;

    // visited array to find the
    // already allocated process
    Boolean []visited = new Boolean[n];
    for (int i = 0; i < n; i++)
    {
        visited[i] = false;
    }

    // work array to store the copy of
    // available resources
    int []work = new int[m];
    for (int i = 0; i < m; i++)
    {
        work[i] = avail[i];
    }

    while (count<n)
    {
        Boolean flag = false;
        for (int i = 0; i < n; i++)
        {
            if (visited[i] == false)
            {
                int j;
                for (j = 0; j < m; j++)
                {
                    if (need[i, j] > work[j])
                    break;
                }
                if (j == m)
                {
                    safeSequence[count++] = i;
                    visited[i] = true;
                    flag = true;
                    for (j = 0; j < m; j++)
                    {
                        work[j] = work[j] + alloc[i, j];
                    }
                }
            }
        }
        if (flag == false)
        {
            break;
        }
    }
    if (count < n)
    {
```

```
        Console.WriteLine("The System is UnSafe!");
    }
    else
    {
        //System.out.println("The given System is Safe");
        Console.WriteLine("Following is the SAFE Sequence");
        for (int i = 0; i < n; i++)
        {
            Console.Write("P" + safeSequence[i]);
            if (i != n - 1)
            Console.Write(" -> ");
        }
    }
}

void calculateNeed()
{
    for (int i = 0;i < n; i++)
    {
        for (int j = 0;j < m; j++)
        {
            need[i, j] = max[i, j] - alloc[i, j];
        }
    }
}

// Driver Code
public static void Main(String[] args)
{
    GFG gfg = new GFG();

    gfg.initializeValues();

    // Calculate the Need Matrix
    gfg.calculateNeed();

    // Check whether system is in
    // safe state or not
    gfg.isSafe();
}
}
```

Output:

Following is the SAFE Sequence
P1 -> P3 -> P4 -> P0 -> P2

LAB TASKS:

Assume that there are 5 processes, P0 through P4, and 4 types of resources. At T0 we have the following system state:

Max Instances of Resource Type A = 3 (2 allocated + 1 Available)
Max Instances of Resource Type B = 17 (12 allocated + 5 Available)
Max Instances of Resource Type C = 16 (14 allocated + 2 Available)
Max Instances of Resource Type D = 12 (12 allocated + 0 Available)

Given Matrices

	Allocation Matrix (N0 of the allocated resources By a process)				**Max Matrix** Max resources that may be used by a process				**Available Matrix** Not Allocated Resources			
	A	**B**	**C**	**D**	**A**	**B**	**C**	**D**	**A**	**B**	**C**	**D**
P_0	0	1	1	0	0	2	1	0	1	5	2	0
P_1	1	2	3	1	1	6	5	2				
P_2	1	3	6	5	2	3	6	6				
P_3	0	6	3	2	0	6	5	2				
P_4	0	0	1	4	0	6	5	6				
Total	**2**	**12**	**14**	**12**								

(a) Create the need matrix (max-allocation)
(b) Use the safety algorithm to test if the system is in a safe state or not?
(c) Write the C# code of the given problem.