SET-224 /CET-225
Operating Systems

LAB # 12

LAB Title

Implementation of Deadlock avoidance mechanism(Banker's Algorithm)

Assessment of CLO: 04, PLO: 03

Student Name:	
Roll No.	
Semester		Session	

S. No.	Perf. Level / Criteria	Excellent (2.5)	Good (2)	Satisfactory (1.5)	Needs Improvement (0 ~ 1)	Marks Obtained
1	Project Execution & Implementation	Fully functional, optimized, and well-structured.	Minor errors, mostly functional.	Some errors, requires guidance.	Major errors, non-functional, or not Performed.	
2	Results & Debugging Or Troubleshooting	Accurate results with effective debugging Or Troubleshooting.	Mostly correct, some debugging Or Troubleshooting needed.	Partial results, minimal debugging Or Troubleshooting.	Incorrect results, no debugging Or Troubleshooting, or not attempted.	
3	Problem-Solving & Adaptability (VIVA)	Creative approach, efficiently solves challenges.	Adapts well, minor struggles.	Some adaptability, needs guidance.	Lacks innovation or no innovation, unable to solve problems.	
4	Report Quality & Documentation	Clear, structured, with detailed visuals.	Mostly clear, minor gaps.	Some clarity issues, missing details.	Poorly structured, lacks clarity, or not submitted.	
	Total Marks Obtained Out of 10					

Experiment evaluated by

Instructor's Name	Engr.Bushra Aziz	
Date		Signature	

Objective

The purpose of this manual is to provide a detailed understanding of the Banker's Algorithm for deadlock avoidance in operating systems. This manual explains the algorithm's concepts, implementation steps, examples, and exercises to reinforce its application. The Banker's Algorithm ensures system stability by preventing processes from entering unsafe states, thus avoiding deadlocks.

Banker's Algorithm for Deadlock Avoidance

Introduction

The Banker's Algorithm is a resource allocation and deadlock avoidance algorithm introduced by Edsger Dijkstra. It is named because it simulates a banker allocating funds to customers in a way that ensures funds are available to meet their maximum demands without leading to a state where further resource allocation could cause a deadlock.

Key Concepts

1. **Available Resources**: The total number of each type of resource currently available in the system.
2. **Maximum Need**: The maximum resources a process may need during its execution.
3. **Allocated Resources**: The resources currently allocated to a process.
4. **Need**: The remaining resources a process needs to complete its execution (calculated as Need = Maximum - Allocation).
5. **Safe State**: A state where the system can allocate resources to all processes in some order without leading to a deadlock.
6. **Unsafe State**: A state where some processes cannot complete execution due to a lack of resources, potentially leading to a deadlock.

Implementation Steps

1. Input the number of processes and resource types.
2. Initialize matrices for **Maximum**, **Allocation**, and **Available** resources.
3. Calculate the **Need** matrix as Need[i][j] = Maximum[i][j] - Allocation[i][j].
4. Check if a request from a process can be safely granted:
    o Ensure the request does not exceed the process's **Need** and the **Available** resources.
    o Temporarily allocate the requested resources and check if the system remains in a safe state.
5. Use the **Safety Algorithm** to determine if the system will remain in a safe state after allocation:
    o Find a process whose **Need** can be satisfied with the **Available** resources.
    o Simulate the process execution by adding its **Allocated** resources back to the **Available** pool.
    o Repeat until all processes can execute or no further allocations can be made.
6. If a safe sequence exists, grant the request; otherwise, deny it.

Example

Input:

Processes: P0, P1, P2, P3, P4

Resource Types: A, B, C

Available: [3, 3, 2]

Lab Experiment 12: **Implementation of Deadlock avoidance mechanism (Banker's Algorithm)**

Maximum:

Process	A	B	C
P0	7	5	3
P1	3	2	2
P2	9	0	2
P3	2	2	2
P4	4	3	3

Allocation:

Process	A	B	C
P0	0	1	0
P1	2	0	0
P2	3	0	2
P3	2	1	1
P4	0	0	2

Step 1: Calculate Need

Process	A	B	C
P0	7	4	3
P1	1	2	2
P2	6	0	0
P3	0	1	1
P4	4	3	1

Step 2: Check Safe State

1. **Available**: [3, 3, 2]
2. Process **P1** can execute (Need ≤ Available):
   o Update **Available**: [5, 3, 2]
3. Process **P3** can execute:
   o Update **Available**: [7, 4, 3]

4. Process **P4** can execute:
   o Update **Available**: [7, 4, 5]
5. Process **P0** can execute:
   o Update **Available**: [7, 5, 5]
6. Process **P2** can execute:
   o Update **Available**: [10, 5, 7]

Safe Sequence: P1 -> P3 -> P4 -> P0 -> P2

Exercises

1. Write a Python program to implement the Banker's Algorithm for the above used matrices.