CET-225

Operating Systems

Experiment # 12

Experiment Title

Implementation of Deadlock avoidance mechanism(Banker's Algorithm)

Assessment of CLO(s): 04

Performed on _____

Student Name:			
Roll No.		**Group**	
Semester		**Session**	

Total (Max)	**Performance (03)**	**Viva (03)**	**File (04)**	**Total (10)**
Marks Obtained				
Remarks (if any)				

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:
    - Ensure the request does not exceed the process's **Need** and the **Available** resources.
    - 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:
    - Find a process whose **Need** can be satisfied with the **Available** resources.
    - Simulate the process execution by adding its **Allocated** resources back to the **Available** pool.
    - 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]
- **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):
    - Update **Available**: [5, 3, 2]
3. Process **P3** can execute:
    - Update **Available**: [7, 4, 3]
4. Process **P4** can execute:
    - Update **Available**: [7, 4, 5]
5. Process **P0** can execute:
    - Update **Available**: [7, 5, 5]
6. Process **P2** can execute:
    - 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.