Bansilal Ramnath Agarwal Charitable Trust's

Vishwakarma Institute of Technology, Pune-37

(An autonomous institute of Savitribai Phule Pune University)

OS Lab 8

Year	Third
Branch	AIDS
Division	AI-A
Batch	3
PRN	12320083
Roll no.	69
Name	Durvesh Chaudhari

PROBLEM STATEMENT

Write a program to check whether given system is in safe state or not using Banker's Deadlock Avoidance algorithm.(must attach single PDF contains description of the Banker's Algorithm , code and output)

The **Banker's Algorithm** is a deadlock avoidance algorithm designed to manage the allocation of resources to multiple processes in a way that prevents deadlock. It was proposed by **Edsger Dijkstra** in 1965, and it's called the Banker's Algorithm because it is analogous to a bank ensuring it has enough resources (money) to satisfy all customer withdrawal requests without running out of funds.

Key Concepts:

1. **Processes and Resources:**
   - There are multiple processes, each of which may need a certain number of resources.
   - There are different types of resources (e.g., memory, CPU, disk space), each available in finite quantities.
2. **Resource Allocation:**
   - Each process may request and hold several instances of each resource type.
   - The Banker's Algorithm allocates resources dynamically to processes but ensures that the system remains in a safe state.

Goal of the Algorithm:

The goal of the Banker's Algorithm is to **ensure that the system never enters a deadlock state** by checking whether granting a resource request leaves the system in a **safe state**. A **safe state** means there exists a sequence in which all processes can finish executing using the available resources without causing a deadlock.

Important Terms:

1. **Available**: The number of available instances of each resource type at any point.
2. **Max**: The maximum demand of each process for each resource type. This indicates how many resources a process may potentially request at any point.
3. **Allocation**: The number of resources of each type that have already been allocated to each process.
4. **Need**: The remaining resource requirements of each process. This can be computed as:
   Need[i]=Max[i]−Allocation[i]
   Where:
   - `Need[i][j]` = The amount of resource `j` that process `i` still needs to complete its task.
5. **Safe State**: A state is considered safe if there is a sequence of processes where each one can finish using the currently available resources or resources freed by previously completed processes.
6. **Unsafe State**: A state is unsafe if there is a potential for deadlock, meaning there's no guarantee that all processes can complete.

Working of the Banker's Algorithm:

1. **Initial Setup:**
   - The system keeps track of the resources allocated to each process (`Allocation`), the maximum resource needs of each process (`Max`), and the available resources (`Available`).
2. **Need Calculation:**
   - The algorithm calculates the `Need` matrix, which represents the remaining resource needs of each process.
3. **Resource Request Handling:**
   - When a process requests resources, the system checks if granting the request will leave the system in a **safe state**.
   - The steps followed when a process requests resources:
     - Check if the requested resources are less than or equal to the process's maximum need.
     - Check if the requested resources are available (i.e., the `Available` pool has enough resources).
     - Temporarily allocate the resources (i.e., pretend the request is granted) and check if the system remains in a safe state using the **Safety Algorithm** (discussed below).
     - If the system is safe, grant the resources. Otherwise, the request is denied.
4. **Safety Algorithm:**
   - This algorithm checks if the system is in a safe state by finding a sequence of processes that can finish with the current available resources.
   - Steps:
     - Find a process whose resource needs can be satisfied with the currently available resources.
     - Simulate the process completion by releasing its allocated resources.
     - Repeat the above steps until all processes are finished or there is no process that can finish.
     - If all processes can finish, the system is in a safe state. Otherwise, it's unsafe.

Program:

```
#include <stdio.h>
#include <stdbool.h>

// Number of processes and resource types
#define P 5 // Number of processes
#define R 3 // Number of resource types

// Function to check if the system is in a safe state
bool isSafe(int processes[], int available[], int max[][R], int
allocation[][R]) {
    int need[P][R]; // Need matrix

    // Calculate the need matrix as Need[i][j] = Max[i][j] -
Allocation[i][j]
```

```c
    for (int i = 0; i < P; i++) {
        for (int j = 0; j < R; j++) {
            need[i][j] = max[i][j] - allocation[i][j];
        }
    }

    bool finish[P] = {false}; // To keep track of finished processes
    int safeSeq[P]; // Safe sequence
    int work[R]; // Work vector to track available resources
    for (int i = 0; i < R; i++) {
        work[i] = available[i];
    }

    int count = 0; // Number of processes in the safe sequence

    // Main logic for checking if the system is in a safe state
    while (count < P) {
        bool found = false;
        for (int p = 0; p < P; p++) {
            // Check if process can be satisfied
            if (!finish[p]) {
                int j;
                for (j = 0; j < R; j++) {
                    if (need[p][j] > work[j]) {
                        break;
                    }
                }
                // If all resources can be allocated to process p
                if (j == R) {
                    for (int k = 0; k < R; k++) {
                        work[k] += allocation[p][k]; // Release
resources
                    }
                    safeSeq[count++] = p;
                    finish[p] = true;
                    found = true;
                }
            }
        }

        // If no process can be allocated resources, the system is
unsafe
        if (!found) {
            printf("System is not in a safe state.\n");
            return false;
        }
    }

    // If the system is safe, print the safe sequence
    printf("System is in a safe state.\nSafe sequence: ");
    for (int i = 0; i < P; i++) {
        printf("%d ", safeSeq[i]);
    }
    printf("\n");

    return true;
```

```
}

int main() {
    int processes[P] = {0, 1, 2, 3, 4}; // Process IDs

    // Available instances of each resource
    int available[R] = {3, 3, 2};

    // Maximum demand of each process
    int max[P][R] = {
        {7, 5, 3},
        {3, 2, 2},
        {9, 0, 2},
        {2, 2, 2},
        {4, 3, 3}
    };

    // Resources allocated to each process
    int allocation[P][R] = {
        {0, 1, 0},
        {2, 0, 0},
        {3, 0, 2},
        {2, 1, 1},
        {0, 0, 2}
    };

    // Check if the system is in a safe state
    isSafe(processes, available, max, allocation);

    return 0;
}
```

Explanation of the Code:

1. **Input Data:**
   - `available[]`: The resources currently available in the system.
   - `max[][]`: The maximum number of resources each process may request.
   - `allocation[][]`: The number of resources currently allocated to each process.
2. **Calculating the Need Matrix:**
   - The `need[][]` matrix is calculated as the difference between the maximum demand (`max[][]`) and the allocated resources (`allocation[][]`).
3. **Safety Algorithm:**
   - The algorithm iterates through each process to check if its remaining resource needs can be satisfied with the currently available resources.
   - If a process can be satisfied, it is added to the safe sequence and its resources are "released" back to the system (i.e., added back to the available resources).
   - If no process can be satisfied in a given iteration, the system is declared unsafe.

4. **Safe Sequence:**
    - If all processes can eventually be satisfied (i.e., all processes finish), the system is in a safe state, and a safe sequence is printed.
    - If a deadlock would occur, the system is declared not in a safe state.

Output: