WEEK – 6: OPERATING SYSTEMS LAB

Aim: Simulate Bankers Algorithm for Dead Lock Avoidance

The Banker's Algorithm is a deadlock avoidance algorithm used in operating systems to manage resources and prevent deadlock situations. It ensures that processes request and release resources in a way that does not lead to a deadlock.

Example of Bankers Algorithm:

How does the Banker's Algorithm work?

The Banker's Algorithm works by maintaining a matrix of resources and processes. Each row represents a process, and each column represents a resource. The matrix contains information about the current state of the system, including the maximum number of resources each process needs, the number of resources currently allocated to each process, and the number of resources available in the system.

The data structure used is:

- Available vector
- Max Matrix
- Allocation Matrix
- Need Matrix

Example: Consider a system that contains five processes P1, P2, P3, P4, P5 and the three resource types A, B and C. Following are the resource types: A has 10, B has 5 and the resource type C has 7 instances.

Process	Allocation			Max			Available		
	A	B	C	A	B	C	A	B	C
P1	0	1	0	7	5	3	3	3	2
P2	2	0	0	3	2	2			
P3	3	0	2	9	0	2			
P4	2	1	1	2	2	2			
P5	0	0	2	4	3	3			

Answer the following questions using the banker's algorithm:

1. What is the reference of the need matrix?
2. Determine if the system is safe or not.
3. What will happen if the resource request (1, 0, 0) for process P1 can the system accept this request immediately?

Ans 2 : The context of the need matrix is as follows:

Need [i] = Max [i] - Allocation [i]

Need for P1: (7, 5, 3) - (0, 1, 0) = 7, 4, 3

Need for P2: (3, 2, 2) - (2, 0, 0) = 1, 2, 2

Need for P3: (9, 0, 2) - (3, 0, 2) = 6, 0, 0

Need for P4: (2, 2, 2) - (2, 1, 1) = 0, 1, 1

Need for P5: (4, 3, 3) - (0, 0, 2) = 4, 3, 1

Process	Need A B C
P1	7 4 3
P2	1 2 2
P3	6 0 0
P4	0 1 1
P5	4 3 1

Hence, we created the context of the need matrix.

Ans. 2: Apply the Banker's Algorithm:

Available Resources of A, B and C are 3, 3, and 2.

Now we check if each type of resource request is available for each process.

Step 1: For Process P1:

Need <= Available

7, 4, 3 <= 3, 3, 2 condition is **false**.

So, we examine another process, P2.

Step 2: For Process P2:

Need <= Available

1, 2, 2 <= 3, 3, 2 condition **true**

New available = available + Allocation

(3, 3, 2) + (2, 0, 0) => 5, 3, 2

Similarly, we examine another process P3.

Step 3: For Process P3:

P3 Need <= Available

6, 0, 0 < = 5, 3, 2 condition is **false**.

Similarly, we examine another process, P4.

Step 4: For Process P4:

P4 Need <= Available

0, 1, 1 <= 5, 3, 2 condition is **true**

New Available resource = Available + Allocation

5, 3, 2 + 2, 1, 1 => 7, 4, 3

Similarly, we examine another process P5.

Step 5: For Process P5:

P5 Need <= Available

4, 3, 1 <= 7, 4, 3 condition is **true**

Newly available resource = Available + Allocation

7, 4, 3 + 0, 0, 2 => 7, 4, 5

Now, we again examine each type of resource request for processes P1 and P3.

Step 6: For Process P1:

P1 Need <= Available

7, 4, 3 <= 7, 4, 5 condition is **true**

New Available Resource = Available + Allocation

7, 4, 5 + 0, 1, 0 => 7, 5, 5

So, we examine another process P2.

Step 7: For Process P3:

P3 Need <= Available

6, 0, 0 <= 7, 5, 5 condition is true

New Available Resource = Available + Allocation

7, 5, 5 + 3, 0, 2 => 10, 5, 7

Hence, we execute the banker's algorithm to find the safe state and the safe sequence like P2, P4, P5, P1 and P3.

Ans. 3: For granting the Request (1, 0, 2), first we have to check that **Request <= Available**, that is (1, 0, 2) <= (3, 3, 2), since the condition is true. So the process P1 gets the request immediately.

Explanation of the Program:

The provided code appears to be an implementation of the Banker's Algorithm in C for deadlock avoidance. This algorithm is used to manage resources and prevent deadlock situations in a multi-process system. Here's an explanation of the code:

1. **Initialization**: The code starts by declaring and initializing various variables and arrays to store information about processes, resources, and their allocation.
2. **Input Gathering**:
    - The number of processes (**n**) and the number of resource classes (**r**) are input from the user.
    - The total available resources for each resource class (**totext**) are also input from the user.
    - The allocated resources for each process and resource class are input and stored in the **resalloc** matrix.
3. **New Request Input**:

- The process making a new request (**p**) is input from the user.
- The requested resources for each resource class are input and stored in the **newreq** array.

4. **Active Processes Input**:
    - Information about processes that are either blocked or running is input from the user. This information is stored in the **block** and **run** arrays.
5. **Resetting Process Status**:
    - The code sets the status of the process making the new request (**p**) to not blocked and not running.
6. **Calculating Total Allocated Resources**:
    - For each resource class, the code calculates the total allocated resources by summing up the allocation for all processes in the **resalloc** matrix. The result is stored in the **totalloc** array.
7. **Determining Active Processes**:
    - The code determines which processes are active (neither blocked nor running) and sets the **active** array accordingly.
8. **Handling the New Request**:
    - The code updates the allocation matrix (**resalloc**) with the new request for the specified process (**p**).
    - It also updates the **totalloc** array to reflect the changes caused by the new request.
9. **Checking for Unsafe State**:
    - The code checks if the new state is unsafe by comparing the total allocated resources (**totalloc**) with the total available resources (**totext**). If any resource class has more allocated resources than available, **u** is set to 1, indicating an unsafe state.

10. **Handling Safe State**:
    - If the state is safe (i.e., **u** is 0), the code proceeds to simulate resource allocation.
    - It starts by copying the current allocated resources into **simalloc**.
    - It then iterates through the active processes to check if they can complete their tasks without causing an unsafe state.
    - If a process can complete its task without causing an unsafe state, it is marked as inactive, and its allocated resources are copied to **simalloc**.
    - If all active processes are checked and none cause an unsafe state, the code concludes that deadlock won't occur (**m** remains 0).
11. **Handling Unsafe State**:
    - If the state is unsafe (i.e., **u** is 1), the code sets the new request as the allocation for process **p**.
    - It also updates the **totalloc** array with the new allocation.
    - The code concludes that deadlock will occur.
12. **Output**:
    - Finally, the code prints whether deadlock will occur or not based on the earlier calculations.
13. **User Interaction**:
    - The code uses **getch()** for user interaction to keep the console window open after displaying the result.

Please note that while this code provides a basic structure for the Banker's Algorithm, there may be errors or limitations in the implementation. It's important to thoroughly test it with various scenarios to ensure its correctness and reliability.

Additionally, the use of **conio.h** functions like **clrscr()** and **getch()** is platform-dependent and may not work on all systems.

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

void main() {
    int n, r, i, j, k, p, u = 0, s = 0, m;
    int block[10], run[10], active[10], newreq[10];
    int max[10][10], resalloc[10][10], resreq[10][10];
    int totalloc[10], totext[10], simalloc[10]; // Fixed variable names



    printf("Enter the no of processes:");
    scanf("%d", &n);
    printf("Enter the no of resource classes:");
    scanf("%d", &r);
    printf("Enter the total existed resource in each class:");
    for (k = 1; k <= r; k++)
        scanf("%d", &totext[k]);
    printf("Enter the allocated resources:");
    for (i = 1; i <= n; i++) {
        for (k = 1; k <= r; k++) {
            scanf("%d", &resalloc[i][k]); // Fixed input to the allocation matrix
```

```
    }
}
printf("Enter the process making the new request:");
scanf("%d", &p);
printf("Enter the requested resource:");
for (k = 1; k <= r; k++)
    scanf("%d", &newreq[k]);
printf("Enter the process which are not blocked or running:");
for (i = 1; i <= n; i++) {
    if (i != p) {
        printf("process %d:\n", i); // Removed +1, it should start from 1
        scanf("%d%d", &block[i], &run[i]);
    }
}
block[p] = 0;
run[p] = 0;
for (k = 1; k <= r; k++) {
    j = 0;
    for (i = 1; i <= n; i++) {
        totalloc[k] = j + resalloc[i][k];
        j = totalloc[k];
    }
}
for (i = 1; i <= n; i++) {
    if (block[i] == 1 || run[i] == 1)
```

```
        active[i] = 1;
    else
        active[i] = 0;
}
for (k = 1; k <= r; k++) {
    resalloc[p][k] += newreq[k];
    totalloc[k] += newreq[k];
}
for (k = 1; k <= r; k++) {
    if (totext[k] - totalloc[k] < 0) {
        u = 1;
        break;
    }
}
if (u == 0) {
    for (k = 1; k <= r; k++)
        simalloc[k] = totalloc[k];
    for (s = 1; s <= n; s++) {
        for (i = 1; i <= n; i++) {
            if (active[i] == 1) {
                j = 0;
                for (k = 1; k <= r; k++) {
                    if ((totext[k] - simalloc[k]) < (max[i][k] - resalloc[i][k])) {
                        j = 1;
                        break;
```

```
                }
            }
            if (j == 0) {
                active[i] = 0;
                for (k = 1; k <= r; k++)
                    simalloc[k] = resalloc[i][k];
            }
        }
    }
    m = 0;
    for (k = 1; k <= r; k++)
        resreq[p][k] = newreq[k];
    printf("Deadlock won't occur");
  } else {
    for (k = 1; k <= r; k++) {
        resalloc[p][k] = newreq[k];
        totalloc[k] = newreq[k];
    }
    printf("Deadlock will occur");
  }
}
```

OUTPUT:

The code you've provided calculates whether a deadlock will occur or not based on the input values and then prints the result.

Input:

Enter the no of processes: 5

Enter the no of resource classes: 3

Enter the total existed resource in each class: 3 3 2

Enter the allocated resources:

0 1 0

2 0 0

3 0 2

2 1 0

0 0 2

Enter the process making the new request: 1

Enter the requested resource: 1 0 2

Enter the process which are not blocked or running:

process 2:

0 1

process 3:

0 1

process 4:

0 1

process 5:

0 1

Sample Output:

Deadlock will occur.

In this sample input, the code calculates that deadlock will occur after processing the request from process 1, and it prints "Deadlock will occur."