**Practical No. 10**

**Aim: Simulate Bankers algorithm for Deadlock Avoidance.**

**Deadlock avoidance**

Deadlock avoidance in operating systems is a proactive strategy aimed at circumventing deadlock scenarios. This approach entails dynamically adjusting system parameters to preemptively steer clear of deadlock situations. Various techniques, such as resource allocation algorithms and process scheduling strategies, are employed to effectively prevent the onset of deadlocks.

**Resource Allocation Graph (RAG):**

The system utilizes a graphical representation to track resource allocation, with processes depicted as nodes and resource types as edges. By analyzing this graph, the system can preemptively identify potential deadlock situations arising from resource allocation. It then employs proactive measures to circumvent these deadlocks, ensuring smooth system operation.

**Banker's Algorithm:**

In this algorithm, processes must declare their maximum resource needs upfront. The system only allocates resources if it determines that it will not lead to unsafe states, where a deadlock could occur.

**Wait-Die and Wound-Wait:**

These are two strategies used in concurrency control to prevent deadlock in multi-threaded systems. In Wait-Die, a younger process waits for an older one to release a resource, while in Wound-Wait, an older process preempts a younger one if it requests a resource it holds.

**Algorithm:**

**Step 1:** Input the number of processes (num\_processes) and the number of resources (num\_resources).

Input the available resources (available), maximum demand matrix (max),

and allocation matrix (allocation) for each process.

Calculate the need matrix (need) as the difference between the maximum demand and allocation for each process.

**Step 2:** Safety Check (is\_safe\_state):

Implement the is\_safe\_state function to determine if the system is in a safe state after allocating resources.

Initialize a work array (work) with the available resources and a finish array

(finish) indicating whether each process has finished.

For the requested resources, check if they exceed the need or available resources.

If so, return false (not safe).

Simulate resource allocation by updating the work and allocation matrices accordingly.

**Step 3:** Use a while loop to iterate until all processes are visited:

Within the loop, iterate over each process and check if its needs can be satisfied with the available resources. If satisfied, update the work array and mark the process as visited.

If a process cannot be satisfied, return false (not safe).

If all processes can be visited, return true (safe).

**Step 4:** Resource Request:

Implement the request\_resources function to handle resource requests from a specified process.

Input the resource request from the user and check if it leads to a safe state using the is safe state function.

If the request is granted (safe), update the available, allocation, and need matrices accordingly. Otherwise, deny the request.

**Step 5:** Main Function:

Input the process for which a resource request is to be made.

Call the request resources function to handle the resource request.

Repeat the process for additional resource requests if needed.

Step 6: End

**Program:-**

#include <stdio.h>

#include <stdbool.h>

#define MAX\_PROCESSES 10

#define MAX\_RESOURCES 10

// Global arrays to store resource allocation information

int available[MAX\_RESOURCES];

int max[MAX\_PROCESSES][MAX\_RESOURCES];

int allocation[MAX\_PROCESSES][MAX\_RESOURCES];

int need[MAX\_PROCESSES][MAX\_RESOURCES];

bool finished[MAX\_PROCESSES];

// Number of processes and resources

int num\_processes, num\_resources;

// Function to check if the system is in a safe state after resource request

bool is\_safe\_state(int process, int request[]) {

int work[MAX\_RESOURCES];

bool finish[MAX\_PROCESSES];

// Initialize work and finish arrays

for (int i = 0; i < num\_resources; i++) {

work[i] = available[i];

}

for (int i = 0; i < num\_processes; i++) {

finish[i] = finished[i];

}

// Try allocating resources

for (int i = 0; i < num\_resources; i++) {

if (request[i] > need[process][i] || request[i] > work[i]) {

return false;

}

}

// Update state after resource allocation

for (int i = 0; i < num\_resources; i++) {

work[i] -= request[i];

allocation[process][i] += request[i];

need[process][i] -= request[i];

}

// Check if the new state is safe

bool visited[num\_processes];

for (int i = 0; i < num\_processes; i++) {

visited[i] = false;

}

int count = 0;

while (count < num\_processes) {

bool found = false;

for (int i = 0; i < num\_processes; i++) {

if (!visited[i]) {

bool satisfied = true;

for (int j = 0; j < num\_resources; j++) {

if (need[i][j] > work[j]) {

satisfied = false;

break;

}

}

if (satisfied) {

for (int j = 0; j < num\_resources; j++) {

work[j] += allocation[i][j];

}

visited[i] = true;

count++;

found = true;

}

}

}

if (!found) {

return false;

}

}

return true;

}

// Function to handle resource request from a process

void request\_resources(int process) {

int request[MAX\_RESOURCES];

// Get resource request from the user

printf("Enter the request for resources from process %d:\n", process);

for (int i = 0; i < num\_resources; i++) {

scanf("%d", &request[i]);

}

// Check if the request can be granted

if (is\_safe\_state(process, request)) {

printf("Request granted.\n");

// Update resource allocation and availability

for (int i = 0; i < num\_resources; i++) {

available[i] -= request[i];

allocation[process][i] += request[i];

need[process][i] -= request[i];

}

} else {

printf("Request denied. Not in a safe state.\n");

}

}

int main() {

// Get input from the user

printf("Enter the number of processes: ");

scanf("%d", &num\_processes);

printf("Enter the number of resources: ");

scanf("%d", &num\_resources);

printf("Enter the available resources:\n");

for (int i = 0; i < num\_resources; i++) {

scanf("%d", &available[i]);

}

printf("Enter the maximum demand matrix:\n");

for (int i = 0; i < num\_processes; i++) {

printf("Process %d: ", i);

for (int j = 0; j < num\_resources; j++) {

scanf("%d", &max[i][j]);

}

}

printf("Enter the allocation matrix:\n");

for (int i = 0; i < num\_processes; i++) {

printf("Process %d: ", i);

for (int j = 0; j < num\_resources; j++) {

scanf("%d", &allocation[i][j]);

need[i][j] = max[i][j] - allocation[i][j];

}

}

// Get the process to request resources for

printf("Enter the process to request resources for: ");

int process;

scanf("%d", &process);

// Call function to handle resource request

request\_resources(process);

return 0;

}

**OUTPUT:**

Enter the number of processes: 2

Enter the number of resources: 3 Enter the available resources:

3

4

2

Enter the maximum demand matrix:

Process 0: 1

0

2

Process 1: 2

1

2

Enter the allocation matrix:

Process 0: 1

3

2

Process 1: 0

1

2

Enter the process to request resources for: 1

Enter the request for resources from process 1:

2

0

Request denied. Not in safe state.