Practical No. 10

Aim: Simulate Bankers algorithm for Deadlock Avoidance.

Deadlock avoidance is a strategy used in operating systems to prevent the occurrence of deadlocks. It involves making decisions dynamically to ensure that the system will not enter into a deadlock state. There are several techniques for deadlock avoidance, including:

Resource Allocation Graph (RAG):

The system maintains a graph that represents the resource allocation state. Processes are represented as nodes, and resource types are represented as edges. The system analyzes this graph to determine if a resource allocation will lead to a deadlock and takes appropriate actions to avoid it.

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.

Dijkstra's Resource Allocation Algorithm:

This algorithm dynamically checks if a resource allocation will lead to a safe state by simulating the allocation process. If the allocation results in an unsafe state, the system rejects it.

These techniques aim to ensure that the system always maintains a safe state, where deadlock cannot occur. However, they may incur overhead in terms of system performance and resource utilization. Therefore, deadlock avoidance strategies must strike a balance between preventing deadlocks and ensuring efficient system operation.

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 for Deadlock avoidance.

```
#include <stdio.h>

#include <stdbool.h>

#define MAX_PROCESSES 10

#define MAX_RESOURCES 10

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];
```

```
int num_processes, num_resources;

bool is_safe_state(int process, int request[]) {

int work[MAX_RESOURCES];

bool finish[num_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;

}

}

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;

}

void request_resources(int process) { int request[MAX_RESOURCES];

printf("Enter the request for resources from process %d:\n", process); for (int i = 0; i < num_resources; i++) { scanf("%d", &request[i]);

}

if (is_safe_state(process, request)) { printf("Request granted.\n"); 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 safe state.\n");

}

}

int main() { 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]; }

}

printf("Enter the process to request resources for: "); int process; scanf("%d", &process);

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.
```