**Practical No. 11**

**Objective:-** **Simulate Bankers Algorithm for deadlock Prevention.**

**Deadlock prevention**

Deadlock prevention in operating systems proactively structures system operations to eliminate the possibility of deadlock occurrences. Unlike deadlock avoidance, which reacts to deadlock situations as they arise, prevention techniques preemptively design system protocols to ensure deadlock cannot manifest.such a way that deadlock becomes impossible.

**Resource Allocation Policies:**

One of the fundamental principles in preventing deadlocks is to design resource allocation policies that do not allow the necessary conditions for deadlock to occur. These policies are typically implemented in the operating system kernel.

Examples of resource allocation policies include the use of preemption, where the operating system can forcibly reclaim resources from processes to resolve potential deadlock situations, or the use of strict resource ordering rules that ensure resources are always requested and released in a consistent order.

By enforcing strict rules and policies governing resource allocation, the system can eliminate the possibility of circular wait conditions, one of the necessary conditions for deadlock.

**Resource Allocation Graph (RAG):**

Another approach to deadlock prevention involves the use of a resource allocation graph (RAG) to represent resource allocation relationships between processes and resources in the system.

In the RAG, processes and resources are represented as nodes, and allocation and request edges connect processes to the resources they hold or are waiting to acquire.

Deadlocks can be prevented by ensuring that the RAG remains free of cycles. This can be achieved by employing algorithms to detect and break cycles in the graph, such as the Banker's algorithm.

By carefully managing resource allocations and monitoring the RAG, the system can prevent deadlocks from occurring.

**Resource Utilization Monitoring:**

Deadlocks can also be prevented by monitoring resource utilization and ensuring that resources are efficiently utilized without leading to resource contention.

Operating system schedulers and resource managers can employ various techniques, such as load balancing and priority-based scheduling, to optimize resource allocation and prevent resource exhaustion or contention.

By actively monitoring resource utilization and dynamically adjusting resource allocation policies, the system can maintain a healthy resource environment and reduce the likelihood of deadlock.

**Timeouts and Deadlock Detection:**

Timeout mechanisms can be incorporated into resource allocation operations to prevent processes from waiting indefinitely for resource requests to be fulfilled.

If a process fails to acquire a requested resource within a specified time period, it may release any currently held resources and retry its request or take alternative actions to avoid deadlock.

Additionally, the operating system may periodically perform deadlock detection checks to identify potential deadlock situations and take preemptive actions to resolve them before they escalate into full-fledged deadlocks.

**Dynamic Resource Management:**

Dynamic resource management techniques involve dynamically adjusting resource allocation policies based on system conditions and workload demands.

By analyzing system resource usage patterns and workload characteristics, the operating system can adapt its resource allocation strategies to prevent resource contention and minimize the risk of deadlock.

Dynamic resource management techniques may include resource reallocation, process prioritization, and adaptive scheduling algorithms that respond to changing system conditions in real-time.

**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.

Initialize the finished array to track the completion status of 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.

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

(finish) indicating whether each process has finished.

**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) .

Resource Request (request\_resources):

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

Input the resource request from the user.

Check if the request exceeds the maximum demand or available resources. If so, deny the request.

If the request is valid, temporarily allocate the resources and check if the system remains in a safe state using the is\_safe\_state function.

If the system remains in a safe state, grant the request. Otherwise, deny the request and undo the resource allocation.

**Step 4:** Main Function:

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

Call the request\_resources function to handle the resource request.

**Step 5:** End

**//Program for deadlock prevention.**

#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];

int num\_processes, num\_resources;

bool is\_safe\_state() {

int work[MAX\_RESOURCES];

bool finish[num\_processes];

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

work[i] = available[i];

}

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

finish[i] = false;

}

int count = 0;

while (count < num\_processes) {

bool found = false;

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

if (!finish[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];

}

finish[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]);

}

bool valid\_request = true;

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

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

valid\_request = false;

break;

}

}

if (valid\_request) {

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

available[i] -= request[i];

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

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

}

if (is\_safe\_state()) {

printf("Request granted.\n");

} else {

printf("Request denied. Not in safe state.\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. Invalid request.\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: 3

Enter the number of resources: 4

Enter the available resources:

2 3 4 1

Enter the maximum demand matrix:

Process 0: 7 5 3 6

Process 1: 3 2 2 2

Process 2: 9 0 2 8

Enter the allocation matrix:

Process 0: 0 1 0 0

Process 1: 2 0 0 0

Process 2: 3 0 2 2

Enter the process to request resources for: 1

Enter the request for resources from process 1:

1 0 2 1

Request denied. Not in safe state.