Experiment no: 8

Name: Saloni Vishwakarma
Batch-Roll no: C1-13
Semester-Section: 4th-C
Subject: OS(Operating System) Lab
Date of execution: 28 June 2023

Aim: Write a C program to demonstrate the Banker's Algorithm and recovery processes.

Theory: The Banker's algorithm is a resource allocation and deadlock avoidance algorithm used in operating systems. It is designed to ensure that resources are allocated in a safe and efficient manner, preventing deadlocks from occurring.

The algorithm is based on the concept of a banker who manages a finite amount of resources and serves multiple processes. Each process has a maximum demand for each resource and can request additional resources as needed. The banker must determine whether granting a request will leave the system in a safe state or potentially lead to a deadlock.

Here's a brief overview of how the Banker's algorithm works:

Initialization: The algorithm starts by determining the total number of resources available and the maximum demand of each process.

Request: When a process requests additional resources, the banker checks if granting the request would result in an unsafe state. If the request can be granted without causing a deadlock, the banker temporarily allocates the resources and reevaluates the system state.

Safety Check: The banker performs a safety check to determine if the system is in a safe state after the resource allocation. This involves simulating the allocation of resources to all processes and checking if there is a sequence that allows each process to complete successfully.

Resource Allocation: If the system is in a safe state, the banker grants the requested resources permanently to the process. If not, the request is denied, and the process must wait until the resources become available.

Release: When a process completes its execution, it releases all allocated resources, making them available for other processes.

The Banker's algorithm ensures that resources are allocated in a way that prevents deadlocks, as it considers the maximum demands of all processes and the availability of resources before granting requests. By maintaining a safe state, the algorithm helps avoid situations where processes are unable to progress due to resource conflicts.

Viva Questions:

1) What is a deadlock?
→ A deadlock is a situation in which two or more processes are unable to proceed because each is waiting for a resource held by another process. It is a state of a system where a process cannot proceed, leading to a permanent halt in the execution of the involved processes.

2) What are the necessary conditions for a deadlock to occur?
→ For a deadlock to occur, the following four conditions, known as the necessary conditions for deadlock, must be present simultaneously: mutual exclusion, hold and wait, no preemption, and circular wait.

3) What is the purpose of the Banker's algorithm?
→ The purpose of the Banker's algorithm is to allocate resources to processes in a safe and efficient manner, while preventing deadlocks.

4) How does the Banker's algorithm help in preventing deadlocks?
→ The Banker's algorithm prevents deadlocks by considering the maximum demands of processes and the availability of resources before granting resource requests. It ensures that a safe state is maintained throughout the resource allocation process.

5) What are the key components required for implementing the Banker's algorithm?
→ The key components required for implementing the Banker's algorithm are the available resources, the maximum resource allocation for each process, the current allocation of resources to processes, and the resource requests made by processes.

6) Explain the concept of a safe state in the context of the Banker's algorithm.
→ In the context of the Banker's algorithm, a safe state is a state in which all processes can complete their execution without causing a deadlock. It means that there is a sequence of resource allocations and deallocations that allows all processes to finish successfully.

Code and Output:

```
#include <stdio.h>
#define MAX_PROCESSES 10
#define MAX_RESOURCES 10

// Function to calculate the need matrix
void calculateNeedMatrix(int need[MAX_PROCESSES][MAX_RESOURCES], int
max[MAX_PROCESSES][MAX_RESOURCES], int
allocation[MAX_PROCESSES][MAX_RESOURCES], int n, int m)
{
   int i,j;
   for ( i = 0; i < n; i++)
   {
      for ( j = 0; j < m; j++)
      {
         need[i][j] = max[i][j] - allocation[i][j];
      }
   }
}

// Function to check if a process can be allocated resources
int isSafe(int available[MAX_RESOURCES], int
need[MAX_PROCESSES][MAX_RESOURCES],int
allocation[MAX_PROCESSES][MAX_RESOURCES], int n, int m)
{
   int work[MAX_RESOURCES];
   int finish[MAX_PROCESSES] = {0};
   int i,k;
   // Initialize work array with available resources
   for ( i = 0; i < m; i++)
   {
      work[i] = available[i];
   }
```

```
    // Find a process that can be executed
    for ( k = 0; k < n; k++)
    {
        for ( i = 0; i < n; i++)
        {
            if (finish[i] == 0)
            {
                int j,r;
                for (j = 0; j < m; j++)
                {
                    if (need[i][j] > work[j])
                    {
                        break;
                    }
                }
                // If all resources for process i can be allocated
                if (j == m)
                {
                    // Allocate resources to process i
                    for ( r = 0; r < m; r++)
                    {
                        work[r] += allocation[i][r];
                    }
                    // Mark process i as finished
                    finish[i] = 1;
                }
            }
        }
    }
   // Check if all processes have finished execution
    for ( i = 0; i < n; i++)
    {
        if (finish[i] == 0) {
            return 0; // System is in an unsafe state
        }
    }
    return 1; // System is in a safe state
}

// Function to display the matrices
```

```
void displayMatrices(int available[MAX_RESOURCES], int
need[MAX_PROCESSES][MAX_RESOURCES], int
allocation[MAX_PROCESSES][MAX_RESOURCES], int n, int m)
{
   int i,j;
   int s[m];
   for(j=0;j<m;j++)
   {
      s[j]=0;
      for(i=0;i<n;i++)
      {
         s[j]=s[j]+allocation[i][j];
      }
   }
   for(i=0;i<m;i++)
   {
      available[i]=available[i]-s[i];
   }
   printf(" AVAILABLE: ");
   for ( i = 0; i < m; i++)
   {
      printf("%d ", available[i]);
   }
   printf("\n\n REMAINING NEED:\n");
   for ( i = 0; i < n; i++)
   {
      printf(" ");
      for ( j = 0; j < m; j++)
      {
         printf("%d ", need[i][j]);
      }
      printf("\n");
   }
   printf("\n PROCESS SEQUENCE: ");
   int work[MAX_RESOURCES];
   int finish[MAX_PROCESSES] = {0};
   // Initialize work array with available resources
   for ( i = 0; i < m; i++)
   {
      work[i] = available[i];
```

```
}
// Find a process that can be executed
int sequence[MAX_PROCESSES];
int seqIndex = 0;
while (seqIndex < n)
{
   int found = 0;
   for ( i = 0; i < n; i++)
   {
      if (finish[i] == 0)
      {
         int r;
         for (j = 0; j < m; j++)
         {
            if (need[i][j] > work[j]) {
               break;
            }
         }
         // If all resources for process i can be allocated
         if (j == m)
         {
            // Allocate resources to process i
            for ( r = 0; r < m; r++)
            {
               work[r] += allocation[i][r];
            }
            // Mark process i as finished
            finish[i] = 1;
            sequence[seqIndex++] = i;
            found = 1;
         }
      }
   }
   // If no process found that can be executed, break the loop
   if (found == 0) {
      break;
   }
}
int k;
// Display process sequence
```

```
    for ( i = 0; i < seqIndex; i++)
    {
       printf("P%d ", sequence[i]);
    }
    printf("\n");
    printf("\n AVAILABLE MATRIX:\n");
    int sum[n][m],p;
    for(i=0;i<n;i++)
    {
      for(k=0;k<m;k++)
      {
         sum[i][k]=available[k]+allocation[sequence[i]][k];
      }
      for(p=0;p<m;p++)
      {
         available[p]=sum[i][p];
      }
    }
    for(i=0;i<n;i++)
    {
       printf(" ");
       for(j=0;j<m;j++)
       {
          printf("%d ",sum[i][j]);
       }
       printf("\n");
    }
}

int main()
{
    int i,j,n, m; // Number of processes and resources
    int available[MAX_RESOURCES];
    int max[MAX_PROCESSES][MAX_RESOURCES];
    int allocation[MAX_PROCESSES][MAX_RESOURCES];
    printf("\n Enter the number of processes: ");
    scanf("%d", &n);
    printf(" Enter the number of resources: ");
    scanf("%d", &m);
    printf(" Enter the number of available resources for each type: ");
```

```
    for ( i = 0; i < m; i++)
    {
       scanf("%d", &available[i]);
    }
    printf("\n Enter the allocation matrix:\n");
    for ( i = 0; i < n; i++)
    {
       printf(" ");
       for ( j = 0; j < m; j++)
       {
          scanf("%d", &allocation[i][j]);
       }
    }
    printf("\n Enter the maximum need matrix:\n");
    for ( i = 0; i < n; i++)
    {
       printf(" ");
       for ( j = 0; j < m; j++)
       {
          scanf("%d", &max[i][j]);
       }
    }
    int need[MAX_PROCESSES][MAX_RESOURCES];
    // Calculate the need matrix
    calculateNeedMatrix(need, max, allocation, n, m);
    // Check if the system is in a safe state
    if (isSafe(available, need, allocation, n, m))
    {
       printf("\n The system is in a safe state.\n");
       displayMatrices(available, need, allocation, n, m);
    }
    else
    {
       printf("\n The system is in an unsafe state. Deadlock may occur.\n");
    }
    return 0;
}
```

```
Enter the number of processes: 5
Enter the number of resources: 3
Enter the number of available resources for each type: 10 5 7

Enter the allocation matrix:
0 1 0
2 0 0
3 0 2
2 1 1
0 0 2

Enter the maximum need matrix:
7 5 3
3 2 2
9 0 2
2 2 2
4 3 3

The system is in a safe state.
AVAILABLE: 3 3 2

REMAINING NEED:
7 4 3
1 2 2
6 0 0
0 1 1
4 3 1

PROCESS SEQUENCE: P1 P3 P4 P0 P2

AVAILABLE MATRIX:
5 3 2
7 4 3
7 4 5
7 5 5
10 5 7
```

Conclusion: We have successfully studied and implemented Banker's algorithm in C. Overall, the Banker's algorithm serves as a valuable tool for resource management and deadlock prevention, contributing to the smooth and reliable operation of modern computer systems.