Text

Description automatically generated

Department of Computer Science & Engineering

November, 2021

**DEADLOCK AVOIDANCE WITH BANKER’S ALGORITHM**

A MINI PROJECT REPORT

*Submitted by:*

Bhat Suhas Sudhir (Reg No: 190905026, Roll No:3, Section: D)

Rishav Raj (Reg No: 190905072, Roll No:7, Section: D)

Naman Aryan (Reg No: 190905080, Roll No:8, Section: D)

Swastik Karwa (Reg No: 190905410, Roll No:53, Section: D)

*In partial fulfillment for the award of the degree of*

B.Tech in Computer Science Engineering

In

Operating System Lab

Department of Computer Science & Engineering

BONAFIDE CERTIFICATE

Certified that this project report “**DEADLOCK AVOIDANCE WITH BANKER’S ALGORITHM**” is the Bonafide work of

Bhat Suhas Sudhir (3,190905026)

Rishav Raj (7,190905072)

Naman Aryan (8,190905080)

Swastik Karwa (53,190905410)

who carried out the mini project work under my supervision.

Dr. Ashalatha Nayak Dr. Ahamed Shafique

Professor and HOD Professor, CSE Department

Submitted to the Viva voce Examination held on 30th of November 2021.

EXAMINER 1 EXAMINER 2

***ACKNOWLEDGEMENT***

*We would like to express my greatest appreciation to all the individuals who have helped and supported me throughout the project. We are thankful to our Operating System teacher for his ongoing support during the project, from initial advice, and encouragement, which led to the final report of this project. I would also like to thank Dr****. Ahamed Shafique*** *and* ***Dr. Praveen Pai T*** *who was always there in our computer lab for assistance*

*A special acknowledgement goes to my classmates who helped me in completing the project by exchanging interesting ideas and sharing their experience.*

*I wish to thank our parents as well for their undivided support and interest who inspired me and encouraged me to go my own way, without whom I would be unable to complete my project.*

*At the end, I want to thank my friends who displayed appreciation to my work and motivated us to continue my work.*

CONTRIBUTION OF EACH STUDENT

* SUHAS BHAT :- REQUIRMENT ANALYSIS AND WRITING CODE .
* NAMAN ARYAN :- CODE ANALYSIS AND REPORT WRITING.
* SWASTIK:- TESTING , DEBUGGING AND CODE EFFICIENCY ANALYSIS.
* RISHAV :- RESEARCH AND REPORT EDITING.

***ABSTRACT***

A Deadlock is a situation where each of the computer process waits for a resource which is being assigned to some another process. In this situation, none of the process gets executed since the resource it needs, is held by some other process which is also waiting for some other resource to be released.

**Strategies for handling Deadlock**

* Deadlock Ignorance. Deadlock Ignorance is the most widely used approach among all the mechanism. ...
* Deadlock prevention. Deadlock happens only when Mutual Exclusion, hold and wait, No preemption and circular wait holds simultaneously. ...
* Deadlock avoidance. ...
* Deadlock detection and recovery.

**Deadlock Avoidance**

In **Deadlock avoidance**, the request for any resource will be granted if the resulting state of the system doesn't cause deadlock in the system. The state of the system will continuously be checked for safe and unsafe states.

In order to, avoid deadlocks, the process must tell OS, the maximum number of resources a process can request to complete its execution.

The simplest and most useful approach states that the process should declare the maximum number of resources of each type it may ever need. The Deadlock avoidance algorithm examines the resource allocations so that there can never be a circular wait condition.

**Input and Output of Bankers Algorithm**

Input of the Banker Algorithm is we take number of processes as input, we take number of resources as input, we take maximum number of resources required by each process of each resource type and also number of allocated resources of each resource type to each process and also number of resources available in the system before allocating resources to the process.

The basic output of the Banker’s Algorithm is to determine if the system is in safe state. If so the safety sequence of the process executed is to be displayed. The outer outputs may also include if a request for resource allocation can be allocated or not.

Also, The Algorithm allows us to check if a process can be executed at a particular iteration.

Bankers Algorithm is implemented by initializing four data structures and analysing it.

Namely,

Let, m = Number of processes and n = Number of resources types

Max matrix (m\*n) : - Indicates the max number of resources required by the process

Allocated matrix (m\*n): - Indicates the number of resources allocated to the each process

Need matrix (m\*n):- Indicates the number or resources currently required by each process.

Available array (size : n):- Indicates the number of resources available in the system.

**Safety Algorithm**

In our version to determine the safety sequence of executing processes we iterate through each process and check if the need of the process is less than the available resources

Need[ith process] [j] < Available[j]

For j = 0 to n

If any process satisfies the above criteria then the process is said to be executed. If none of the process could be executed in an iteration then the system is said to be unsafe.

**We have added two more features to the regular banker’s algorithm:**

1) Resource Request Algorithm

2) Intermediate Process Check

**Resource-Request Algorithm**

If any processes, at a particular moment has any urgent need to be serviced with few instances of resources, then this algorithm helps the system determine if the request can be serviced or denied.

The criteria to check if a request of ith process can be serviced or not:

1) Request[j] < need[i][j]

2) Request[j] <available[j]

If both the criteria are satisfied, then only the request for the resource is serviced else its denied.

**Intermediate Process Check**

In our version of the Banker’s algorithm, in every iteration the checking for process that can be executed starts from the first process to last process in that order respectively. If any specific process has to be checked for its possibility of execution in a particular iteration then this functionality can be used.

**Learning outcome-**

By implementing this project, we learnt about deadlock and the situations in which it arises. We also learnt how to prevent deadlock and if it occurs, what methods can be used to handle it. We learnt how to keep the system in a safe state, and if possible, how to service the request of a process by allocating resources to it without entering unsafe or deadlock state. We learnt about the various ways to execute processes sequentially so that the system never enters an unsafe state.

***Implementation (Code)***

#include <stdio.h>

#include <stdlib.h>

/\*\*

\* Method to print the system is safe and also prints the safety sequnce

\* @param safetySequence [Integer 1D Array] Safety sequnce is stored in the array in order of first safe process to execute to last process executed

\* @param n [Integer] Number of processes OR number of elements in the complete array

\*/

void safe(int \*safetySequence, int n)

{

    int i;

    printf("\nThe given system is safe.\n");

    printf("The Safe Sequence is : ");

    printf("<\t");

    for(i = 0; i < n; i++)

    {

        printf("%d\t",safetySequence[i] + 1); //Prints safety sequence

    }

    printf(">\n");

}

/\*\*

\* Method to print the system is unsafe and also prints the sequnce of processes successfully executed before becoming unsafe

\* @param safetySequence [Integer 1D Array] Sequnce of processes executed is stored in the array in order of first safe process to execute

\* to last process executed

\* @param n [Integer] Number of processes safely executed OR number of elements in the complete array

\*/

void unsafe(int \*safetySequence, int n)

{

    int i;

    printf("\nThe given system is unsafe.\n");

    printf("The Sequence of processes executed before entering unsafe state is : ");

    printf("<\t");

    for(i = 0; i < n; i++)

    {

        printf("%d\t",safetySequence[i] + 1); //Prints processes that successfully executed before entering unsafe state

    }

    printf(">\n");

}

/\*\*

\* Function to allocate resources. i.e 2D Array to max, allocated, and need (with number of rows equal to number of processes and

\* number of columns equal to number of resource types), also allocates 1D array to avaialbe (with number of elements equal to number of resource types)

\* @param max [Pointer to Integer Pointer] Indicates max resources required for all the processes

\* @param allocated [Pointer to Integer Pointer] Indicates allocated resources for all the processes

\* @param need [Pointer to Integer Pointer] Indicates number of resources still required for all the processes

\* @param available [Integer Pointer] Indicates number of available resources

\* @param nProcesses [Integer] Number of Processes

\* @param nResources [Integer] Number of Resource types

\*/

void allocateResources (int \*\*\* max, int \*\*\* allocated, int \*\*\* need, int \*\* available, int nProcesses, int nResources)

{

    int i;

    if(nProcesses <= 0 || nResources <= 0) //If number of processes or number of resource types is negative or zero, throw error

    {

        printf("Error: Invalid number of Processes or Resource types\n");

        exit(EXIT\_FAILURE);

    }

    \*max = (int \*\*)calloc(nProcesses,sizeof(int \*)); //Allocating rows (= Number of processes) to max

    \*allocated = (int \*\*)calloc(nProcesses,sizeof(int \*)); //Allocating rows (= Number of processes) to allocated

    \*need = (int \*\*)calloc(nProcesses,sizeof(int \*)); //Allocating rows (= Number of processes) to need

    \*available = (int \*)calloc(nResources,sizeof(int)); //Allocating 1D Array with size equal number of resource types to available

    if(max == NULL || allocated == NULL || need == NULL || available == NULL) //If any of the above allocation failed, throw error

    {

        printf("Error: Resource allocation failed\n");

        exit(EXIT\_FAILURE);

    }

    for(i = 0; i < nProcesses; i++)

    {

        (\*max)[i] = (int \*)calloc(nResources,sizeof(int)); //Allocating columns (= Number of resource types) to i th row of max

        (\*allocated)[i] = (int \*)calloc(nResources,sizeof(int)); //Allocating columns (= Number of resource types) to i th row of allocated

        (\*need)[i] = (int \*)calloc(nResources,sizeof(int)); //Allocating columns (= Number of resource types) to i th row of need

        if((\*max)[i] == NULL || (\*allocated)[i] == NULL || (\*need)[i] == NULL) //If any of the above allocation failed, throw error

        {

            printf("Error: Resource allocation failed\n");

            exit(EXIT\_FAILURE);

        }

    }

}

/\*\*

\* Function to initialize 2D matrix of max, indicating max resources of each resource type required by a process,

\* This method directly takes input from the user and intializes the matrix.

\* @param max [2D Matrix (size : nProcesses \* nResources)] Indicates max resources required for all the processes

\* @param nProcesses [Integer] Number of Processes

\* @param nResources [Integer] Number of Resource types

\*/

void inputMax(int \*\* max, int nProcesses, int nResources)

{

    int i,j;

    printf("\nEnter the Maximum number of Resources of each Resource type for each process\n");

    for(i = 0; i < nProcesses; i++)

    {

        printf("\nEnter Maximum number of instance of Resources required for Process %d\n",i+1);

        for(j = 0; j < nResources; j++)

        {

            printf("Enter maximum number of intances of Resource %d required by Process %d : \t",j + 1, i + 1);

            scanf("%d",&(max[i][j])); // Inputs max resources for (i + 1)th process and (j + 1)th resource in max[i][j]

            if (max[i][j] < 0) //If max resources is less than zero, throw error

            {

                printf("Error: Invalid number of Resources entered\n");

                exit(EXIT\_FAILURE);

            }

        }

    }

}

/\*\*

\* Function to initialize 2D matrix of allocated, indicating number of resources of each resource type already allocated to the process,

\* This method directly takes input from the user and intializes the matrix.

\* @param max [2D Matrix (size : nProcesses \* nResources)] Indicates max resources required for all the processes

\* @param allocated [2D Matrix (size : nProcesses \* nResources)] Indicates allocated resources for all the processes

\* @param nProcesses [Integer] Number of Processes

\* @param nResources [Integer] Number of Resource types

\*/

void inputAllocated(int \*\* max, int \*\* allocated, int nProcesses, int nResources)

{

    int i,j;

    printf("\nEnter number of Resources of each Resource type already allocated to each process\n");

    for(i = 0; i < nProcesses; i++)

    {

        printf("\nEnter number of instances of Resources allocated for Process %d\n",i+1);

        for(j = 0; j < nResources; j++)

        {

            printf("Enter number of intances of Resource %d already allocated to Process %d : \t",j + 1, i + 1);

            scanf("%d",&(allocated[i][j])); // Inputs allocated resources for (i + 1)th process and (j + 1)th resource in allocated[i][j]

            /\* If allocated resources is greater than max

             required resources OR allocated resources

             is less than zero, throw error   \*/

            if (allocated[i][j] > max[i][j] || allocated[i][j] < 0)

            {

                printf("Error: Invalid number of Resources entered\n");

                exit(EXIT\_FAILURE);

            }

        }

    }

}

/\*\*

\* Function to initialize 2D matrix of need, indicating number of resources of each resource still needed by the process to complete

\* its execution, This method computes the need matrix by subtracting max matrix with allocated matrix. The resulting matrix is need matrix

\* @param max [2D Matrix (size : nProcesses \* nResources)] Indicates max resources required for all the processes

\* @param allocated [2D Matrix (size : nProcesses \* nResources)] Indicates allocated resources for all the processes

\* @param need [2D Matrix (size : nProcesses \* nResources)] Indicates number of resources still required for all the processes

\* @param nProcesses [Integer] Number of Processes

\* @param nResources [Integer] Number of Resource types

\*/

void computeNeed(int \*\* max, int \*\* allocated, int \*\* need, int nProcesses, int nResources)

{

    int i,j;

    for(i = 0; i < nProcesses; i++)

    {

        for(j = 0; j < nResources; j++)

        {

            /\* Need of (i + 1)th max number of (j + 1)th numbrer of already allocated

            process for (j + 1)th = resource required by (i + 1)th - (j + 1)th resource type to

            resource is process (i + 1)th process \*/

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

        }

    }

}

/\*\*

\* Function to initialize 1D array of available, indicating available resources of each type in the system. This method inputs total number

\* resources in the system and then computes available by subtracting the total resources by sum of resources allocated to all processes.

\* @param allocated [2D Matrix (size : nProcesses \* nResources)] Indicates allocated resources for all the processes

\* @param available [1D Array (size : nProcesses)] Indicates number of available resources

\* @param nProcesses [Integer] Number of Processes

\* @param nResources [Integer] Number of Resource types

\*/

void inputAndComputeAvailable(int \*\* allocated, int \* available, int nProcesses, int nResources)

{

    int \* totalAllocated;

    int \* totalResources;

    int i,j;

    totalAllocated = (int \*)calloc(nResources,sizeof(int)); //Allocating 1D Array with size equal number of resource types

    totalResources = (int \*)calloc(nResources,sizeof(int)); //Allocating 1D Array with size equal number of resource types

    for(i = 0; i < nResources; i++)

    {

        totalAllocated[i] = 0; //Initializing totalAllocated array to zero

    }

    for(i = 0; i < nProcesses; i++)

    {

        for(j = 0; j < nResources; j++)

        {

            totalAllocated[j] = totalAllocated[j] + allocated[i][j]; //Computing total allocated Resources of each type

        }

    }

    printf("\nEnter Total Resources (before allocation) in the system\n");

    for(i = 0; i < nResources; i++)

    {

        printf("Enter number of intances of Resource %d : \t",i + 1);

        scanf("%d",&(totalResources[i])); //Inputs total instances of the (i + 1)th Resource in the system before allocation

        /\*If total Resources in the System is less than total allocated

        Resources OR if total Resources is less than zero, throw error \*/

        if(totalResources[i] < totalAllocated[i] || totalResources[i] < 0)

        {

            printf("Error: Invalid number of Resources entered\n");

            exit(EXIT\_FAILURE);

        }

        /\* Number of available Total number of (i + 1)th Total number of allocated

         instances of (i + 1)th = Resource in the system - resources of (i + 1)th

         resource before allocation type to all processes \*/

        available[i] = totalResources[i] - totalAllocated[i];

    }

    free(totalResources); //Frees memory allocated to totalResources array

    free(totalAllocated); //Frees memory allocated to totalAllocated array

}

/\*\*

\* Function to initialize resources. i.e. Max resources required by the process, Number of Allocated resources to the process,

\* Number of resources needed by the process, and Number of available resources

\* @param max [2D Matrix (size : nProcesses \* nResources)] Indicates max resources required for all the processes

\* @param allocated [2D Matrix (size : nProcesses \* nResources)] Indicates allocated resources for all the processes

\* @param need [2D Matrix (size : nProcesses \* nResources)] Indicates number of resources still required for all the processes

\* @param available [1D Array (size : nProcesses)] Indicates number of available resources

\* @param nProcesses [Integer] Number of Processes

\* @param nResources [Integer] Number of Resource types

\*/

void initializeResources(int \*\* max, int \*\* allocated, int \*\* need, int \* available, int nProcesses, int nResources)

{

    inputMax(max,nProcesses,nResources); //Inputs Max requried resources and intializes max matrix

    inputAllocated(max,allocated,nProcesses,nResources); //Inputs Allocated resources and intializes allocated matrix

    computeNeed(max,allocated,need,nProcesses,nResources); //Computes Need from Max and allocated and intializes need matrix

    inputAndComputeAvailable(allocated,available,nProcesses,nResources); /\*Inputs total Resources in the System and compute available resources

from it. And thus, intializes available array \*/

}

/\*\*

\* Frees memory of all the important data structues to which memory was allocated.

\* @param max [2D Matrix (size : nProcesses \* nResources)] Indicates max resources required for all the processes

\* @param allocated [2D Matrix (size : nProcesses \* nResources)] Indicates allocated resources for all the processes

\* @param need [2D Matrix (size : nProcesses \* nResources)] Indicates number of resources still required for all the processes

\* @param available [1D Array (size : nProcesses)] Indicates number of available resources

\* @param nProcesses [Integer] Number of Processes

\*/

void freeResources(int \*\* max, int \*\* allocated, int \*\* need, int \* available, int nProcesses)

{

    int i;

    for(i = 0; i < nProcesses; i++) //Frees ith row of max, allocated, need matrix

    {

        free(max[i]);

        free(allocated[i]);

        free(need[i]);

    }

    free(available); //Frees memory allocated to available array

    /\*Frees memory allocated to max, allocated and need which

    are now essentially an array of integer pointers\*/

    free(max);

    free(allocated);

    free(need);

}

/\*\*

\* Does an iteration over the process to see which process's need can be satisfied with the available resources and can be executed.

\* @param need [2D Matrix (size : nProcesses \* nResources)] Indicates number of resources still required for all the processes

\* @param available [1D Array (size : nProcesses)] Indicates number of available resources

\* @param nProcesses [Integer] Number of Processes

\* @param nResources [Integer] Number of Resource types

\* @return [Integer] Returns the process number/id that can be safely executed OR returns -1 if no process could be executed and indicates unsafe state

\*/

int iterativeCheck(int \*\* need, int \* available, int nProcesses, int nResources)

{

    int i,j,flag;

    for(i = 0; i < nProcesses; i++) //Iterating over all process

   {

        flag = 0; //Initializing flag to 0

        /\*If the process has been already executed or if a process need for a

        certain resource is greater than the availability of the resource,

        set the flag to one indicating the current process can't be executed in this cycle.

        i.e.for ith process if need[i][j] <= available[j] for all j = 1 to nResources\*/

        for(j = 0; j < nResources; j++)

        {

            if(need[i][j] == -1 || need[i][j] > available[j])

            {

                flag = 1;

            }

        }

        if(flag == 0) //If flag is still 0, this process is safe to execute, hence return process id/number

        {

            return i;

        }

    }

    return -1; //If we reach here, it indicates that none of the process are safe to execute and system is unsafe, thus return -1

}

/\*\*

\* This method frees allocated resources of a given process and stores it in available indicating the resources available for other processes.

\* This process also updates the need table for the process to -1 indicating the process has completed its execution

\* @param allocated [2D Matrix (size : nProcesses \* nResources)] Indicates allocated resources for all the processes

\* @param need [2D Matrix (size : nProcesses \* nResources)] Indicates number of resources still required for all the processes

\* @param available [1D Array (size : nProcesses)] Indicates number of available resources

\* @param nResources [Integer] Number of Resource types

\* @param processNo The process number for which the allocated resources need to be freed

\*/

void freeAllocated (int \*\* allocated, int \*\* need, int \*available, int nResources, int processNo)

{

   int i;

   for(i = 0; i < nResources; i++)

   {

       need[processNo][i] = -1; //Need of the completed process is updated to -1 which indicates completion

       available[i] = available[i] + allocated[processNo][i]; //Makes the allocated resouces available for other processes

       allocated[processNo][i] = 0; //Updates the allocated resources to zero

   }

}

/\*\*

\* User interface to allow urgent request and process check

\* @param interationNo [Integer] Denotes the current iteration / Number of processes completed

\* @return [Integer] returns the choice entered by the user

\*/

int UI(int iterationNo)

{

    int choice;

    choice = 0;

    while(!(choice >= 1 && choice <=3)) //Loop until you get a valid choice from user

    {

        printf("\nIteration No : %d\n",iterationNo);

        printf("1)Do an iteration\n");

        printf("2)Urgent Request\n");

        printf("3)Check process\n");

        printf("Enter your choice : \t");

        scanf("%d",&choice);

        if(!(choice >= 1 && choice <=3))

        {

            printf("\nInvalid choice! Try Again.\n");

        }

    }

    return choice;

}

/\*\*

\* Checks if urgent request of resourcesby any process can be served, (i.e. Requested resource is < available resource)

\* if yes, then the request is served else request is declined

\* @param allocated [2D Matrix (size : nProcesses \* nResources)] Indicates allocated resources for all the processes

\* @param need [2D Matrix (size : nProcesses \* nResources)] Indicates number of resources still required for all the processes

\* @param available [1D Array (size : nProcesses)] Indicates number of available resources

\* @param nProcesses [Integer] Number of Processes

\* @param nResources [Integer] Number of Resource types

\*/

void urgentRequest(int \*\* allocated, int \*\* need, int \* available, int nProcesses, int nResources)

{

    int processNo,i;

    int \* urgentResource;

    printf("Enter the process number which has made the urgent request");

    scanf("%d",&processNo);

    processNo = processNo - 1;

    if(processNo < 0 || processNo >= nProcesses) //If invalid process number, deny urgent request

    {

        printf("Invalid process number. Urgent Request denied\n");

        return;

    }

    if(need[processNo][0] == -1) //If process already executed, deny urgent request

    {

        printf("Process already executed. Urgent Request denied\n");

        return;

    }

    urgentResource = (int \*) calloc(nResources,sizeof(int));

    for(i = 0; i < nResources; i++) //Input the number of instances of resources urgently required

    {

        printf("Enter instances of Resource %d that is urgently required\n", i + 1);

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

        if(urgentResource[i] + allocated[processNo][i] > need[processNo][i]) //If requested instances are more than the need of the process, deny the request

        {

            printf("Invalid Request, more than the need of the process, Request denied\n");

            free(urgentResource);

            return;

        }

        if(urgentResource[i] > available[i]) //If requested instances are more than available instances, deny the request

        {

            printf("Request cannot be satisfied due to unavailability of Resources. Request denied\n");

            free(urgentResource);

            return;

        }

    }

    /\*If program reaches her it means the request can be satisfied, hence increase

    allocated resources and reduce the available resources and need of the proces\*/

    for(i = 0; i < nResources; i++)

    {

        allocated[processNo][i] = allocated[processNo][i] + urgentResource[i]; //Increasing the allocated resource by requested number of resources

        need[processNo][i] = need[processNo][i] - urgentResource[i]; //Reducing the need of the process by the requested resources

        available[i] = available[i] - urgentResource[i]; //Reducing the available instances by the requested resources

    }

    printf("Request can be satisfied and hence requested resources are allocated\n");

    free(urgentResource);

}

/\*\*

\* Checks if the inputed process number can be executed sucessfully given the current circumstances.

\* @param need [2D Matrix (size : nProcesses \* nResources)] Indicates number of resources still required for all the processes

\* @param available [1D Array (size : nProcesses)] Indicates number of available resources

\* @param nProcesses [Integer] Number of Processes

\* @param nResources [Integer] Number of Resource types

\* @return [Integer] The function returns the process number if it is sucessfully executed, -1 if the process is already executed or if the

\* available resources is less than the needed resources.

\*/

int processCheck(int \*\* need, int \* available, int nProcesses, int nResources)

{

    int processNo, flag, i;

    printf("\nEnter the process number of the process to be checked :\t");

    scanf("%d",&processNo);

    processNo = processNo -1;

    if(processNo < 0 || processNo >= nProcesses) //If invalid process number, return -1

    {

        printf("Invalid process number.\n");

        return -1;

    }

    if(need[processNo][0] == -1) //If process already executed, return -1

    {

        printf("Process already executed.\n");

        return -1;

    }

    flag = 0;

    // Loop over every resource type to check if the process's every resource types's need can be satisfied by the availble resources

    for(i = 0; i < nResources; i++)

    {

        if(need[processNo][i] > available[i])

        {

            flag = 1;

        }

    }

    /\*If flag stays 0 at the end of loop, it indicates that the process can be satisfied by the available resources, hence return

    process number so that the process can be executed\*/

    if(flag == 0)

    {

        return processNo;

    }

    /\*If flag changes to 1, one or more resource type's avaialble instances cannot satisfy the need of the process, hence

    process cannot be executed and return -1 \*/

    else

    {

        printf("The process cannot be executed as available instances of Resources cannot satisfy the need of the process\n");

        return -1;

    }

}

/\*\*

\* Executes banker's algorithm and determines if the sytem is in the safe state or not.

\* @param allocated [2D Matrix (size : nProcesses \* nResources)] Indicates allocated resources for all the processes

\* @param need [2D Matrix (size : nProcesses \* nResources)] Indicates number of resources still required for all the processes

\* @param available [1D Array (size : nProcesses)] Indicates number of available resources

\* @param nProcesses [Integer] Number of Processes

\* @param nResources [Integer] Number of Resource types

\*/

void bankersAlgo(int \*\* allocated, int \*\* need, int \* available, int nProcesses, int nResources)

{

    int processNo, nCompleted, choice;

    int \* safetySequence;

    nCompleted = 0; //Initializes number of completed processes to zero

    safetySequence = (int \*)calloc(nProcesses, sizeof(int)); //Allocates an 1D Array to store the safety sequence

    while(nCompleted != nProcesses)

    {

        choice = UI(nCompleted);

        if(choice == 1)

        {

            processNo = iterativeCheck(need,available,nProcesses,nResources); /\*Call to iterativeCheck checks which process can be executed safely

                                                                                avoiding deadlock. If no process could be run, returns -1\*/

        }

        if(choice == 2)

        {

            urgentRequest(allocated,need,available,nProcesses,nResources); // Checks if urgent request of resources by any process can be served

            continue;

        }

        if(choice == 3)

        {

            processNo = processCheck(need,available,nProcesses,nResources); /\*Checks if prcess could be executed, if yes, returns the processNo, if

                                                                             no, returns -1 \*/

            if(processNo == -1)

            {

                continue;

            }

        }

        /\*If processNo is -1, it means no process in the current iteration

         could be executed and hence call to unsafe prints the system is unsafe and

        processes executed until the moment and exits \*/

        if (processNo == -1)

        {

            unsafe(safetySequence,nCompleted);

            free(safetySequence);

            return;

        }

        else

        {

            freeAllocated(allocated,need,available,nResources,processNo); /\*Frees the resources allocated to the process that just finished

                                                                         execution and makes it available for the other processes.

                                                                         updates need of the executed process to -1, indicating completion\*/

            printf("Process %d Completed its execution\n",processNo + 1);

            safetySequence[nCompleted] = processNo; //Stores the process's id number in the safety sequence

            nCompleted++;                                     //Incrementing number of completed processes

        }

    }

    safe(safetySequence,nCompleted); /\*The algorithm reaches here only if all the processes are complete, hence the call to safe

                                                 prints that the system is safe and the safety sequecne\*/

    free(safetySequence);

}

/\*\*

\* Main function, takes number of input and number of resources, allocates the required data structures and calls banker's algorithm

\* @return [Integer] Returns EXIT\_SUCCESS(0) on sucessfully execution, else returns something else

\*/

int main()

{

    int nProcesses,nResources;

    int \*\* need;

    int \*\* max;

    int \*\* allocated;

    int \* available;

    int \* safetySequence;

    printf("Enter number of Processes\n");

    scanf("%d",&nProcesses); //Inputs number of processes and stores it in nProcesses

    printf("Enter number of Resource types\n");

    scanf("%d",&nResources); //Inputs number of resource types and stores it in nResources

    allocateResources(&max,&allocated,&need,&available,nProcesses,nResources); /\* Calls allocateResources to allocate resources to

     need, max, allocated and available \*/

    initializeResources(max,allocated,need,available,nProcesses,nResources); /\* Calls initializeResources to intialize all the

                                                                                required data structures \*/

    bankersAlgo(allocated,need,available,nProcesses,nResources); /\* Executes banker's algorithm \*/

    freeResources(max,allocated,need,available,nProcesses); /\* Frees memory from all the allocated resources back

                                                                                the system \*/

    return EXIT\_SUCCESS;

}

***Hardware / programming languages :-***

We have used the Ubuntu operating system to develop the software for the project. The code has been written and developed in the C language.

***Learning outcome:-***

By implementing this project, we learnt about deadlock and the situations in which it arises. We also learnt how to prevent deadlock and if it occurs, what methods can be used to handle it. We learnt how to keep the system in a safe state, and if possible, how to service the request of a process by allocating resources to it without entering unsafe or deadlock state. We learnt about the various ways to execute processes sequentially so that the system never enters an unsafe state.

***Future Scope :-***

1. Can be used to develop a better OS where possibility of deadlock is minimized.
2. Can be used for processes that uses threads and depends on multiple resources.
3. Can be used to determine the order of execution of the processes within the process.