# Laboratory 7

Title of the Laboratory Exercise: Programs for deadlock avoidance algorithm

1. Introduction and Purpose of Experiment

Deadlocks can be avoided if certain information is available in advance. By solving these problems students will become familiar to avoid deadlock in advance with the available resource information

1. Aim and Objectives

Aim

* To develop Bankers algorithm for multiple resources for deadlock avoidance

Objectives

At the end of this lab, the student will be able to

* Verify a problem to check that whether deadlock will happen or not for the given resources
* Implement the banker’s algorithm for multiple resources

1. Experimental Procedure
   * 1. Analyse the problem statement
     2. Design an algorithm for the given problem statement and develop a flowchart/pseudo-code
     3. Implement the algorithm in C language
     4. Compile the C program
     5. Test the implemented program
     6. Document the Results
     7. Analyse and discuss the outcomes of your experiment
2. Questions

Implement a Bankers algorithm for deadlock avoidance

1. Calculations/Computations/Algorithms

STEP 1: Start

STEP 2: initialize requestMatrix, finished (array), availCopy(array) and count 0

STEP 3: for every row in request matrix, do

3.1: for every col in row, do

3.1.1: requestMatrix[row][col] = maxRequired[row][col] – currentAlloc[row][col]

STEP 4: availCopy = copy of availableVector

STEP 5: while count is less than number of processes, do

5.1: isSafe false

5.2: for every process p, do

5.2.1: if p is not finished, do

5.2.1.1: if p can be completed with current available resources, free resources, add p to sequence, mark as finished, isSafe true

5.3: if isSafe is false, return false

STEP 6: return true

STEP 7: Stop

1. Presentation of Results

#include <stdio.h>

#include <stdbool.h>

#define NUM\_PROCESSES 5

#define NUM\_RESOURCE\_CLASSES 3

bool isStateSafe(int[], int[], int[][NUM\_RESOURCE\_CLASSES], int[][NUM\_RESOURCE\_CLASSES], int[]);

int main(int *argc*, char \**argv*[])

{

*// defining variable to store the process ids*

    int processes[] = {0, 1, 2, 3, 4};

*// Available instances of resources*

    int avail[] = {3, 3, 2};

*// Maximum R that can be allocated*

*// to processes*

    int maxm[][NUM\_RESOURCE\_CLASSES] = {{7, 5, 3},

                                        {3, 2, 2},

                                        {9, 0, 2},

                                        {2, 2, 2},

                                        {4, 3, 3}};

*// Resources allocated to processes*

    int allot[][NUM\_RESOURCE\_CLASSES] = {{0, 1, 0},

                                         {2, 0, 0},

                                         {3, 0, 2},

                                         {2, 1, 1},

                                         {0, 0, 2}};

*// declaring variable to store processes sequence*

    int seq[NUM\_PROCESSES];

    bool res = isStateSafe(processes, avail, maxm, allot, seq);

    if (res)

    {

        printf("This state is safe. The sequence that leads to completetion of all processes is:\n");

        for (int i = 0; i < NUM\_PROCESSES; i++)

            printf("%d ", seq[i]);

        printf("\n");

    }

}

*/\*\**

*\* Checks if the given state is a safe state or not.*

*\* @param processes the process ids.*

*\* @param availableVector the vector containing the amount of resources available per class.*

*\* @param maxRequired contains the maximum number of resources required to complete*

*\* (for each process).*

*\* @param currentAlloc contains the number of resources held by each process.*

*\* @param seq to store the sequence of processes that need to be executed that lead to*

*\* completion.*

*\* @return true, if state is safe, false otherwise.*

*\*/*

bool isStateSafe(int *processes*[], int *availableVector*[], int *maxRequired*[][NUM\_RESOURCE\_CLASSES],

                 int *currentAlloc*[][NUM\_RESOURCE\_CLASSES], int *seq*[])

{

*// declaring request matrix*

    int requestMatrix[NUM\_PROCESSES][NUM\_RESOURCE\_CLASSES];

*// initializing all processes as not finished*

    bool finished[NUM\_PROCESSES] = {0};

*// variable to store copy of availableVector*

    int availCopy[NUM\_RESOURCE\_CLASSES];

*// variable to keep track of loop*

    int count = 0;

*// finding the request matrix*

    for (int i = 0; i < NUM\_PROCESSES; i++)

        for (int j = 0; j < NUM\_RESOURCE\_CLASSES; j++)

            requestMatrix[i][j] = *maxRequired*[i][j] - *currentAlloc*[i][j];

*// creating copy*

    for (int i = 0; i < NUM\_RESOURCE\_CLASSES; i++)

        availCopy[i] = *availableVector*[i];

*// while all processes are not finished*

    while (count < NUM\_PROCESSES)

    {

        bool isSafe = false;

*// for every process, do*

        for (int i = 0; i < NUM\_PROCESSES; i++)

        {

*// if process is not finished, do*

            if (finished[i] == 0)

            {

                int j;

*// checking if process i can be completed with the*

*// available resources*

                for (j = 0; j < NUM\_RESOURCE\_CLASSES; j++)

                    if (requestMatrix[i][j] > availCopy[j])

*// process cannot be completed, break out of loop*

                        break;

*// if it was able to be completed, do*

                if (j == NUM\_RESOURCE\_CLASSES)

                {

*// freeing resources*

                    for (int k = 0; k < NUM\_RESOURCE\_CLASSES; k++)

                        availCopy[k] += *currentAlloc*[i][k];

*// adding process to sequence*

*seq*[count++] = *processes*[i];

*// marking process as finished*

                    finished[i] = 1;

*// a safe state is found*

                    isSafe = true;

                }

            }

        }

*// if no safe state was found, return false*

        if (isSafe == false)

            return false;

    }

    return true;

}

**Figure 1 : Source Code**

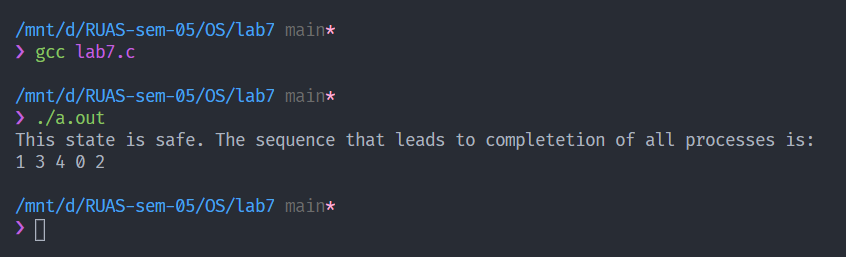


Figure 2: Execution

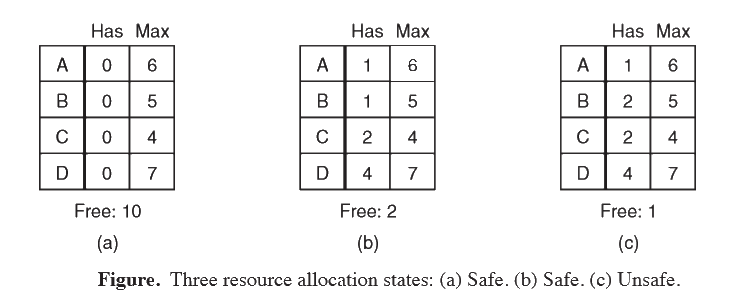
1. Analysis and Discussions

The banker's algorithm is a scheduling algorithm that can avoid deadlocks.

* It considers each request as it occurs.
* If granting the request leads to a safe state (ie., a state where even if one process asks for all the available resources, it can complete its execution), it grants it.
* If granting the request leads to unsafe state, it does not grant it.

Assume that there are four processes, *A, B, C* and *D*, and a single resource of 32 instances in total.

* Process *A* required 6 instances to complete.
* Process *B* required 5 instances to complete.
* Process *C* required 4 instances to complete.
* Process *D* required 7 instances to complete.



The following allocation:

* 1 to *A*,
* 1 to *B*,
* 2 to *C*, and
* 4 to *D*

is safe, since using the remaining 2 instances, process *C* can finish execution. However, the allocation,

* 1 to *A*,
* 2 to *B*,
* 2 to *C*, and
* 4 to *D*

is unsafe, as using the 1 remaining instance, no process can be completed.

The banker's algorithm for a single resource can be generalized to handle multiple resources. The generalized version closely resembles the deadlock detection with multiple resources of each type algorithm.



The matrix on the left is the current allocation matrix, and the matrix on the right is the request matrix. *E* is the existing resource vector, *P* is the processed resources (resources that are assigned), and *A* is the available resource vector.

Like the Banker's algorithm for a single resource, a request for a resource will be granted only if granting it leads to a safe state.

1. Conclusions

Banker’s algorithm is an algorithm that is used to avoid deadlocks. This algorithm was introduced by Dijkstra. This algorithm can be used only if the number of resources that a process needs is known beforehand. This is usually not the case, as a process does not always know when it will need a resource.

1. Comments

1. Limitations of Experiments

The algorithm can only be used if the number of resources that are required by a process is known beforehand, that is, the maximum allocation matrix is known before the process starts execution.

2. Limitations of Results

The program has a time complexity of roughly where is the number of processes, and is the number of resources classes. Performing this algorithm every time a resource is requested can be expensive.

3. Learning happened

The Banker’s algorithm for multiple resources was learned.