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

2. 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

3. Experimental Procedure

   i. Analyse the problem statement
   ii. Design an algorithm for the given problem statement and develop a flowchart/pseudo-code
   iii. Implement the algorithm in C language
   iv. Compile the C program
   v. Test the implemented program
   vi. Document the Results
   vii. Analyse and discuss the outcomes of your experiment

4. Questions

   Implement a Bankers algorithm for deadlock avoidance

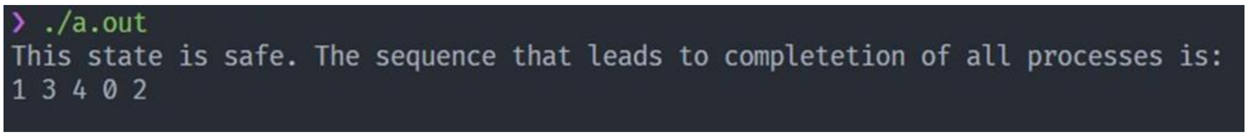
5. Calculations/Computations/Algorithms

```
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
```

6. 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_C
LASSES], int[]);

int main(int argc, char *argv[])
{
int processes[] = {0, 1, 2, 3, 4};
int avail[] = {3, 3, 2};
int maxm[][NUM_RESOURCE_CLASSES] = {{7, 5, 3},
{3, 2, 2},
{9, 0, 2},
{2, 2, 2},
{4, 3, 3}};

int allot[][NUM_RESOURCE_CLASSES] = {{0, 1, 0},
{2, 0, 0},
{3, 0, 2},
{2, 1, 1},
{0, 0, 2}};

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 p
rocesse  s 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 availab
le per class
.
*   @param maxRequired contains the maximum number of resources required to comp
lete
*   (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_R
ESOURCE_CLA SSES],
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 (isSafe == false) return false;
}

return true;
}
```

Output of the program:-



```
❯ ./a.out
This state is safe. The sequence that leads to completetion of all processes is:
1 3 4 0 2
```

7. Analysis and Discussions

It is a banker algorithm used to avoid deadlock and allocate resources safely to each process in the computer system. The 'S-State' examines all possible tests or activities before deciding whether the allocation should be allowed to each process. It also helps the operating system to successfully share the resources between all the processes. The banker's algorithm is named because it checks whether a person should be sanctioned a loan amount or not to help the bank system safely simulate allocation resources. In this section, we will learn the Banker's Algorithm in detail. Also, we will solve problems based on the Banker's Algorithm. To understand the Banker's Algorithm first we will see a real word example of it.

Suppose the number of account holders in a particular bank is 'n', and the total money in a bank is 'T'. If an account holder applies for a loan; first, the bank subtracts the loan amount from full cash and then estimates the cash difference is greater than T to approve the loan amount. These steps are taken because if another person applies for a loan or withdraws some amount from the bank, it helps the bank manage and operate all things without any restriction in the functionality of the banking system.

Similarly, it works in an operating system. When a new process is created in a computer system, the process must provide all types of information to the operating system like upcoming processes, requests for their resources, counting them, and delays. Based on these criteria, the operating system decides which process sequence should be executed or waited so that no deadlock occurs in a system. Therefore, it is also known as deadlock avoidance algorithm or deadlock detection in the operating system.

8. 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.

9. 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 $O(pr)$where $p$is the number of processes, and r 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.