TRIBHUVAN UNIVERSITY
INSTITUTE OF ENGINEERING
THAPATHALI CAMPUS

A LAB REPORT ON

Banker's Algorithm for Deadlock Avoidance

SUBMITTED BY
Sushovan Shakya
THA075BEI046

SUBMITTED TO
Department of Electronics and Computer Engineering

26th August, 2021

BANKER'S ALGORITHM FOR DEADLOCK AVOIDANCE

OBJECTIVE

- To implement Banker's algorithm.

THEORY

The banker's algorithm is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation for predetermined maximum possible amounts of all resources, then makes an "s-state" check to test for possible activities, before deciding whether allocation should be allowed to continue.

The name was chosen because the algorithm could be used in a banking system to ensure that the bank never allocated its available cash in such a way that it could no longer satisfy the needs of all its customers.

Several data structures must be maintained to implement the banker's algorithm. These data structures encode the state of the resource-allocation system.

Available: A vector of length m indicates the number of available resources of each type. If Available[j] equals k, then k instances of resource type RR_{jj} are available.

Max: An n × m matrix defines the maximum demand of each process. If Max[i][j] equals k, then process P_{i} may request at most k instances of resource type RR_{jj}.

Allocation: An *n* × *m* matrix defines the number of resources of each type currently allocated to each process. If Allocation[i][j] equals *k,* then process PP_{ii} is currently allocated *k* instances of resource type RR_{jj}.

Need: An n × m matrix indicates the remaining resource need of each thread. If $NNNNNNNN[ii][jj]$ equals k, then process P_{i} may need k more instances of resource type RR_{jj} to complete its task. Note that

$NNNNNNNN[ii][jj] = MMMMMM[ii][jj] - AAAAAAAAAAMMAAiiAAAA[ii][jj]$.

Safety Algorithm

It is the algorithm for finding out whether or not a system is in a safe state.

This algorithm can be described as follows:

1. Let Work and Finish be vectors of length m and n, respectively. Initialize Work = Available and Finish[i] = false for i = 0, 1, ..., n – 1.
2. Find an index *i* such that both
    i. Finish[*i*] == false
    ii. Need[*i*] ≤ Work

   If no such *i* exists, go to step 4.

3. Work = Work + Allocation[*i*] Finish[*i*] = true Go to step 2.
4. If Finish[*i*] == true for all *i*, then the system is in a safe state.

This algorithm may require an order of $mm \times AA^2$ operations to determine whether a state is safe.

Resource-Request Algorithm

It is the algorithm for determining whether requests can be safely granted.

Let $RRNNRRRRNNRRAA_{ii}$ be the request vector for process PP_{ii}. If $RRNNRRRRNNRRAA_{ii}[jj] == kk$, then process P_i wants k instances of resource type RR_{jj}. When a request for resources is made by process P_i, the following actions are taken:

1. If $RRNNRRRRNNRRAA_{ii} \leq NNNNNNNN_{ii}$, go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.
2. If $RRNNRRRRNNRRAA_{ii} \leq AAAAMMiiAAMMAAAANN$, go to step 3. Otherwise, PP_{ii} must wait, since the resources are not available.
3. Have the system pretend to have allocated the requested resources to thread PP_{ii} by modifying the state as follows:

$$AAAAMMiiAAMMAAAANN = AAAAMMiiAAMMAAAANN - RRNNRRRRNNRRAA_{ii}$$

$$AAAAAAAAAAMMAAiiAAAA_{ii} = AAAAAAAAAAMMAAiiAAAA_{ii} + RRNNRRRRNNRRAA_{ii}$$

$$NNNNNNNN_{ii} = NNNNNNNN_{ii} - RRNNRRRRNNRRAA_{ii}$$

If the resulting resource-allocation state is safe, the transaction is completed, and process PP_{ii} is allocated its resources. However, if the new state is unsafe, then PP_{ii} must wait for $RRNNRRRRNNRRAA_{ii}$, and the old resource-allocation state is restored.

Source Code

```
// Banker's Algorithm
#include <stdio.h>
int main()
{
    // P0, P1, P2, P3, P4 are the Process names here

    int n, m, i, j, k;
    n = 5; // Number of processes
    m = 3; // Number of resources

    // Allocation Matrix
    int alloc[5][3] = {
            { 0, 1, 0 }, // P0
            { 2, 0, 0 }, // P1
            { 3, 0, 2 }, // P2
            { 2, 1, 1 }, // P3
            { 0, 0, 2 }  // P4
     };

    // MAX Matrix
    int max[5][3] = {
        { 7, 5, 3 },  // P0
        { 3, 2, 2 },  // P1
        { 9, 0, 2 },  // P2
        { 2, 2, 2 },  // P3
        { 4, 3, 3 }   // P4
    };

    int avail[3] = { 3, 3, 2 }; // Available Resources

    int f[n], ans[n], ind = 0;
    for (k = 0; k < n; k++) {
        f[k] = 0;
    }
    int need[n][m];
    for (i = 0; i < n; i++) {
        for (j = 0; j < m; j++)
            need[i][j] = max[i][j] - alloc[i][j];
    }
    int y = 0;
    for (k = 0; k < 5; k++) {
        for (i = 0; i < n; i++) {
            if (f[i] == 0) {

                int flag = 0;
                for (j = 0; j < m; j++) {
                    if (need[i][j] > avail[j]){
                        flag = 1;
                         break;
                    }
                }

                if (flag == 0) {
                    ans[ind++] = i;
```

```
                    for (y = 0; y < m; y++)
                        avail[y] += alloc[i][y];
                    f[i] = 1;
                }
            }
        }
    }

    printf("Following is the SAFE Sequence\n");
    for (i = 0; i < n - 1; i++)
        printf(" P%d ->", ans[i]);
    printf(" P%d\n\n", ans[n - 1]);

    return (0);
}
```

Output

```
kachila@pop-os:~/Desktop/os_lab/Lab 5$ ./bankers
Following is the SAFE Sequence
 P1 -> P3 -> P4 -> P0 -> P2
```

DISCUSSION AND CONCLUSION

Hence, in this lab session we studied and understood the concept behind Banker's algorithm for deadlock avoidance. The algorithm was implemented using C programming language with necessary data structures for matrices and vectors.