**LOVELY PROFESSIONAL UNIVERSITY**

**Faculty of Technology and Sciences**

**School of Computer Science and Engineering**

**Name of the faculty member:** Priya Virdi

**Course Code:** CSE 316

**Course Title:** Operating System

**Student Name:** Amit Pratap Singh Chauhan

**Student ID: 11705361**

**Email Address: amitchauhank17hv@gmail.com**

**GitHub Link-** <https://github.com/Amit-Pratap-Singh-Chauhan/os-project>

**Code:**

#include<stdio.h>

#include<stdlib.h>

#include<string.h>

#include<stdbool.h>

const int P = 5;

const int R = 4;

void calculateNeed(int need[P][R],int max[P][R],int alloc[P][R])

{

for (int i = 0; i < P; i++)

for (int j = 0; j < R ; j++)

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

}

bool isSafe(int processes[], int avail[], int max[][R],

int alloc[][R])

{

int need[P][R];

calculateNeed(need, max, alloc);

bool finish[6] = {0};

int safeSeq[P];

int work[R];

for (int i = 0; i < R ; i++)

work[i] = avail[i];

int count = 0;

while (count < P)

{

bool found = false;

for (int p = 0; p < P; p++)

{

if (finish[p] == 0)

{

int j;

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

if (need[p][j] > work[j])

break;

if (j == R)

{

for (int k = 0 ; k < R ; k++)

work[k] += alloc[p][k];

safeSeq[count++] = p;

finish[p] = 1;

found = true;

}

}

}

if (found == false)

{

printf("System is not in safe state");

return false;

}

}

printf("Congratulations.....System is in safe state.\nSafe Sequence is: ");

for (int i = 0; i < P ; i++)

printf("%d \n",safeSeq[i]);

return true;

}

int main()

{

printf("Welcome..This Programme works on the concept of Banker's Algorithm.\n");

printf("In this we will find out the safe sequence for the System.\n");

printf ("\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\n");

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

int avail[] = {1, 5, 2, 0};

int max[6][8] = {{0, 0, 1, 2},

{1, 7, 5, 0},

{2, 3, 5, 6},

{0, 6, 5, 2},

{0, 6, 5, 6}};

int alloc[5][8] = {{0, 0, 1, 2},

{1, 0, 0, 0},

{1, 3, 5, 4},

{0, 6, 3, 2},

{0, 0, 1, 4}};

isSafe(processes, avail, max, alloc);

return 0;

}

**EXPLANATION:** The Banker algorithm, sometimes referred to as the detection algorithm, is a resource allocation and deadlock avoidance algorithm that tests for safety by simulating the allocation of predetermined maximum possible amounts of all resources, and then makes an safe-state check to test for possible deadlock conditions for all other pending activities, before deciding whether allocation should be allowed to continue.

When a new process enters a system, it must declare the maximum number of instances of each resource type that it may ever claim; clearly, that number may not exceed the total number of resources in the system. Also, when a process gets all its requested resources it must return them in a finite amount of time.

**ALGORITHM:** Following Data structures are used to implement the Banker’s Algorithm:

Let **‘n’**be the number of processes in the system and **‘m’**be the number of resources types.

**Available:**

* It is a 1-d array of size **‘m’** indicating the number of available resources of each type.
* Available[ j ] = k means there are **‘k’** instances of resource type **Rj**

**Max:**

* It is a 2-d array of size ‘**n\*m’**that defines the maximum demand of each process in a system.
* Max[ i, j ] = k means process **Pi** may request at most **‘k’** instances of resource type **Rj**

**Allocation:**

* It is a 2-d array of size**‘n\*m’**that defines the number of resources of each type currently allocated to each process.
* Allocation[ i, j ] = k means process **Pi** is currently allocated **‘k’** instances of resource type **Rj**

**Need:**

* It is a 2-d array of size **‘n\*m’** that indicates the remaining resource need of each process.
* Need [ i,   j ] = k means process **Pi** currently need **‘k’** instances of resource type **Rj**

For its execution.

* Need [ i,   j ] = Max [ i,   j ] – Allocation [ i,   j ]

Allocation specifies the resources currently allocated to process Pi and Need specifies the additional resources that process Pi may still request to complete its task.

Banker’s algorithm consist of Safety algorithm and Resource request algorithm.

**Safety Algorithm**

The algorithm for finding out whether or not a system is in a safe state can be described as follows:

1) Let Work and Finish be vectors of length ‘m’ and ‘n’ respectively.  
Initialize: Work = Available  
Finish[i] = false; for i=1, 2, 3, 4….n

2) Find an i such that both  
a) Finish[i] = false  
b) Needi <= Work  
if no such i exists goto step (4)

3) Work = Work + Allocation[i]  
Finish[i] = true  
goto step (2)

4) If Finish [i] = true for all i  
then the system is in a safe state

**Resource-Request Algorithm**

Let Requesti be the request array for process Pi. Requesti[j] = k means process Pi wants k instances of resource type Rj. When a request for resources is made by process Pi, the following actions are taken:

1) If Requesti <= Needi  
Goto step (2); otherwise, raise an error condition, since the process has exceeded its maximum claim.

2) If Requesti <= Available  
Goto step (3); otherwise, Pi must wait, since the resources are not available.

3) Have the system pretend to have allocated the requested resources to process Pi by modifying the state as  
follows:  
Available = Available – Requesti  
Allocationi = Allocationi + Requesti  
Needi = Needi– Request

**COMPLEXITITY:** The time complexity of the Banker's algorithm as a function of the number n of processes and m of resources is O(n\*n\*m).

**Limitations of the Banker’s Algorithm:**

1. It is time consuming to execute on every request of each resource.
2. If the claim information [information about resource] is not accurate, system resources may be underutilized.
3. When a system is heavily loaded, very few safe sequences remain as so many resources are granted.
4. Arrival of new process may create problems:
5. The requested resources by the process must be less than the total number of available resources.
6. Since the state without the new process is safe, it would be also safe with the new process. The new process will be added at the end.
7. To avoid starvation problem among processes it require a little more work but it not much harder.
8. If resource becomes unavailable, it can result in an unsafe state.

**BOUNDARY CONDITIONS:**

Boundary conditions for this code is: For Process P is 5 which are P0,P1,P2,P3 and P5 and for Resource R is 4 which are A,B,C,D.

**TEST CASE:**

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Available | | | | Processes | Allocation | | | | Max | | | |
| A | B | C | D | A | B | C | D | A | B | C | D |
| 1 | 5 | 2 | 0 | P0 | 0 | 0 | 1 | 2 | 0 | 0 | 1 | 2 |
|  | | | | P1 | 1 | 0 | 0 | 0 | 1 | 7 | 5 | 0 |
| P2 | 1 | 3 | 5 | 4 | 2 | 3 | 5 | 6 |
| P3 | 0 | 6 | 3 | 2 | 0 | 6 | 5 | 2 |
| P4 | 0 | 0 | 1 | 4 | 0 | 6 | 5 | 6 |

**Safe Sequence for this case is P0 P2 P3 P4 P1.**