**LOVELY PROFESSIONAL UNIVERSITY**  
**Academic Task-3 (Operating System) Compulsory Component**

School of Computer Science and Engineering Faculty of Technology And Sciences

**Name of the faculty** **member**: Isha

**Course Code**: CSE 316 **Course Title**: Operating System

**Term**: Spring term (2019-2020) **Date of Submission**: 12th April 2020

**Student Name: IRLAPATI PAAVANKUMAR**

**Student ID : 11803029**

**Section : K18JC**

**ROLL NO: B42**

**Email Address : ipk2679@gmail.com**

**GitHub Link :** <https://github.com/Paavankumar2001/system-safe-state-detection>

**1.Question :19.** There are 5 processes and 3 resource types, resource A with 10 instances, B with 5 instances and C with 7 instances. Consider following and write a c code to find whether the system is in safe state or not?

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Available | | | Processes | Allocation | | | Max | | |
| A | B | C | A | B | C | A | B | C |
| 3 | 3 | 2 | P0 | 0 | 1 | 0 | 7 | 5 | 3 |
|  | | | P1 | 2 | 0 | 0 | 3 | 2 | 2 |
| P2 | 3 | 0 | 2 | 9 | 0 | 2 |
| P3 | 2 | 1 | 1 | 2 | 2 | 2 |
| P4 | 0 | 0 | 2 | 4 | 3 | 3 |

**Code:**

// Banker's Algorithm

#include <stdio.h>

int main()

{

// P0, P1, P2, P3, P4 are the Process names here

int m, n, i, j, k;

m = 5; // Number of processes

n = 3; // Number of resources

int allocated[5][3] = { { 0, 1, 0 }, // P0 // Allocation Matrix

{ 2, 0, 0 }, // P1

{ 3, 0, 2 }, // P2

{ 2, 1, 1 }, // P3

{ 0, 0, 2 } }; // P4

int max\_needed[5][3] = { { 7, 5, 3 }, // P0 // MAX Matrix

{ 3, 2, 2 }, // P1

{ 9, 0, 2 }, // P2

{ 2, 2, 2 }, // P3

{ 4, 3, 3 } }; // P4

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

int initial[m], ans[m], ind = 0;

for (i = 0; i < m; i++)

{

initial[i] = 0;

}

int rem\_needed[m][n];

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

{

for (k = 0; k < n; k++)

rem\_needed[j][k] = max\_needed[j][k] - allocated[j][k];

}

int y = 0;

for (i = 0; i < 5; i++)

{

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

{

if (initial[j] == 0)

{

int flag = 0;

for (k = 0; k < n; k++)

{

if (rem\_needed[j][k] > available[k])

{

flag = 1;

break;

}

}

if (flag == 0)

{

ans[ind++] = j;

for (y = 0; y < m; y++)

available[y] += allocated[j][y];

initial[j] = 1;

}

}

}

}

printf("The SAFE Sequence is\n");

for (j = 0; j < m - 1; j++)

printf(" P%d ->", ans[j]);

printf(" P%d", ans[m - 1]);

return (0);

}

**1. Description:**

The total number of processes given are said to be p0, p1, p2, p3, p4, and the resources are A, B, C.

For resource A it is having in total 10 instances and for B, C in total 5, 7 instances respectively.

For each and every process the 3 resources comes into work/usage at a time and the process runs only if the current process resources are having enough instances satisfying the maximum needed instances; which refers even if a single resource not having enough instance w.r.t max needed, the whole process cannot run.

According to banker’s algorithm each process skips to the next process if the process is not having enough instances and this sequence repeats among the processes until for a certain process the required resources satisfies the available resource instances so that some resource instance can be released for the those processes whose requirement to the max needed instances of a resource hasn’t been satisfied earlier can now run as there will be more available instances.

And it can be said as an deadlock situation if the maximum needed instances are greater than the total instances. I.e., (max needed) > (total instances) = deadlock.

And this sequence runs until each and every process runs.

The total available resource instances at the end of the sequence = total instances of each resources.

If the problem/situation satisfies each and every condition mentioned above, it is said to be in safe state.

The safe state refers to nothing but not having any deadlock situation.

**2.Agorithm:**

**Banker's algorithm**

When a new process enters the system, it must declare the maximum number of instances of each resource type that it may need. This number may not exceed the total number of resources in the system. When a user requests a set of resources, the system must determine whether the allocation of these resources will leave the system in a safe state. If it will, the resources are allocated; otherwise, the process must wait until some other process releases enough resources.

The data structures encode the state of the resource-allocation system. Here n is the number of processes in the system and m is the number of resource types:

**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 rj are available.

**Max needed:** an n x m matrix defines the maximum demand of each process. If max[i] [j] equals k, then process pi may request at most k instances of resource type rj.

**Allocation:** an n x m matrix defines the number of resources of each type currently allocated to each process. If allocation[i][j] equals k, then process pi is currently allocated k instances of resource type rj.

**Needed**: an n x m matrix indicates the remaining resource need of each process. If need[i][j] equals k, then process pi may need k more instances of resource type ri to complete its task. Note that need[i][j] equals max[i][j] - allocation [i][j].

**Safety algorithm**

For if a system is in a safe state the following algorithm is used

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.

1. Find an index i such that both
2. Finish[i] == False
3. Need ≤ Work

If no such i exists, go to step 4.

1. Work = Work + Allocation;

Finish[i] = True

go to step 2.

1. If Finish[i] == True for all i, then the system is in a safe state.

This algorithm may require an order of m x n2 operations to determine whether a state is safe.

**Resource-request algorithm**

For requests to be safely granted. Let Requesti be the request vector for process pi.

If Requesti [j] == k, then process pi wants k instances of resource type rj. When a request for resources is made by process pi, the following actions are taken:

If Requesti ≤ needi, go to step 2. Otherwise, raise an error condition, since the process has exceeded its maximum claim.

If requesti ≤ available, go to step 3. Otherwise, pi must wait, since the resources are not available.

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 ;

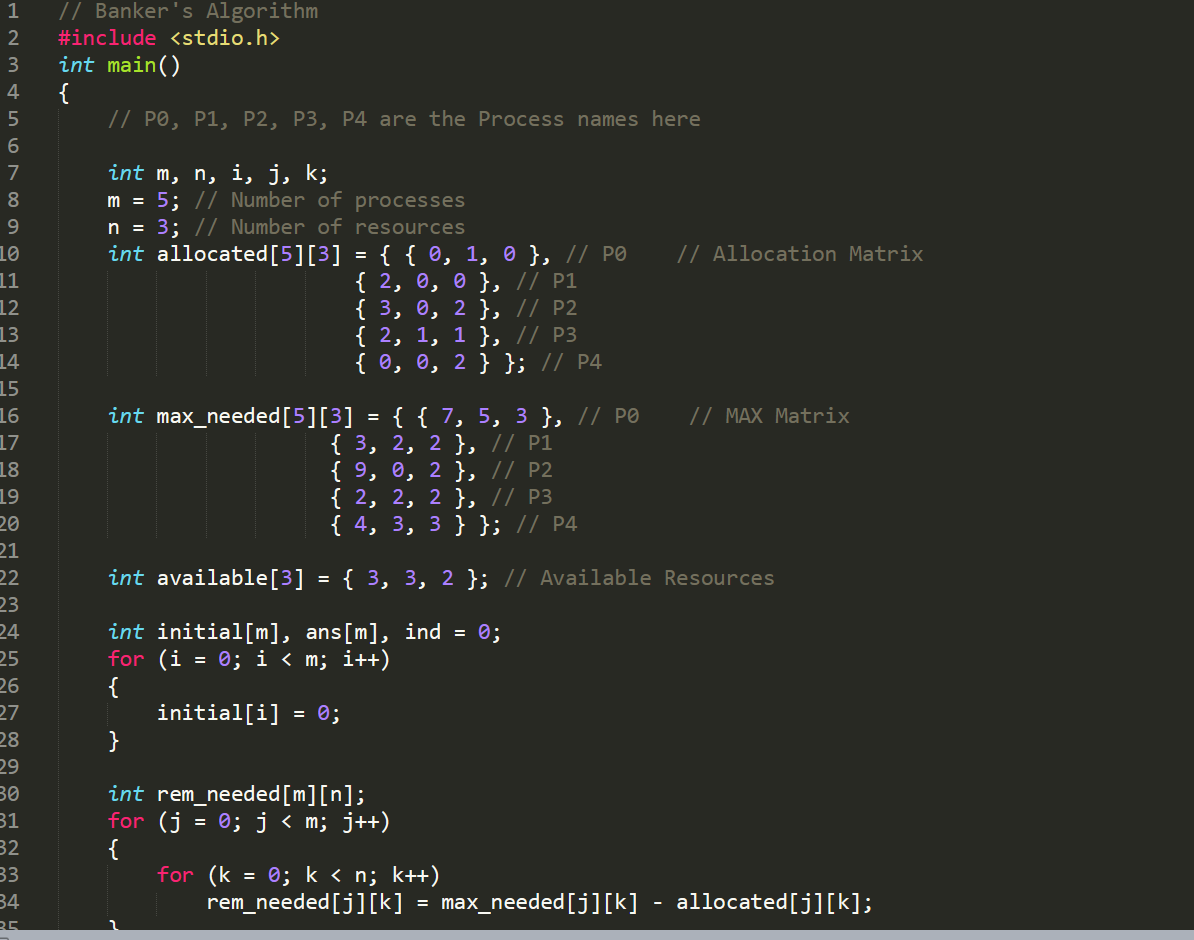
Needi = Needi - Requesti ;

If the resulting resource-allocation state is safe, the transaction is completed, and process pi is allocated its resources. However, if the new state is unsafe, then pi must wait for requesti, and the old resource-allocation state is restored.

3. **Complexity:**

**Description (purpose of use):**

**Safety algorithms complexity is m\*n2**

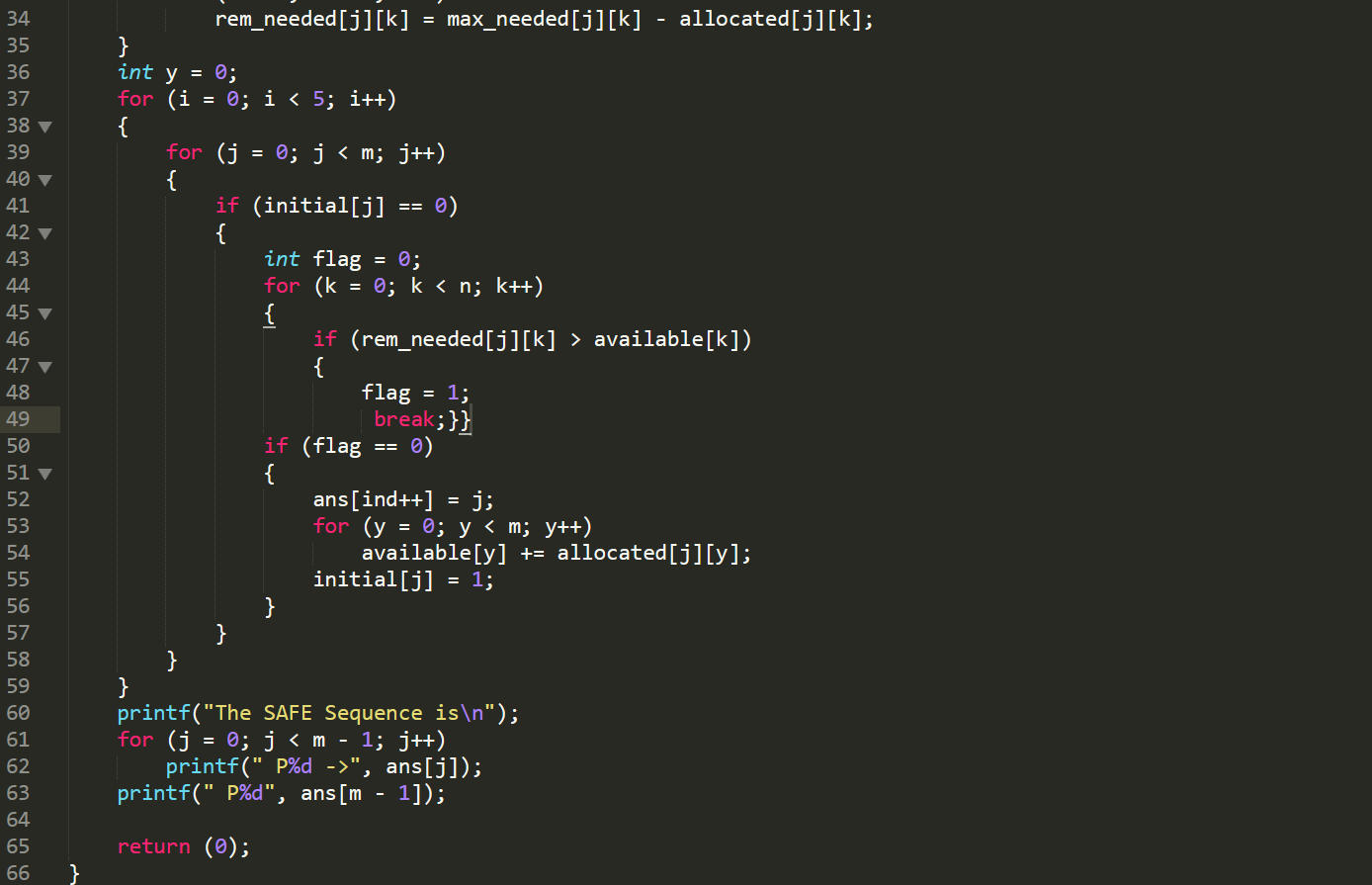
1. **Code snippet:** 

Line 10 to 14 defines the allocated instances for each resource in each process.

Line 16 to 20 defines the maximum instance required for the process w.r.t their resources.

Line 22 the available instances currently.

Line 25 to 28 defines the code for avoiding the garbage values.

Line 30 to 35 defines the code for calculating the remaining needed instances for the resources to run their respective processes.

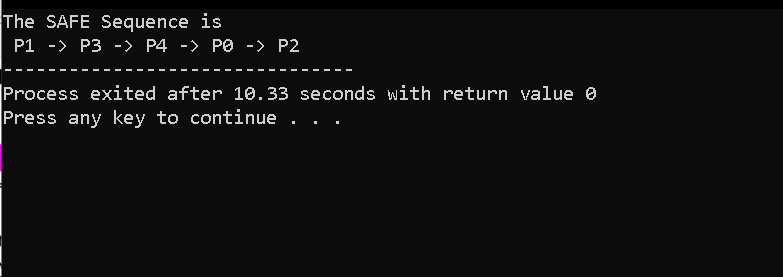
Line 37 to 59 the code which contains the safety algorithm

Line 43 to 49 calculates and tests if a certain process has enough instances for their maximum needed to the resources, and it continues to run the process if it satisfies the condition. And if not flag = 0, which defines to skip to the next process in the sequence.

Line 50 to 55 defines on allocation and adding up to the available instances so that whose process which has not run yet would get the needed instances as, once a process runs the held resources will be released.

Line 61 to 63 defines in printing the sequence.

1. Explain the boundary conditions of the implemented code.



1. **Test cases**

**Description:**

1. for initial state m = 5, n = 3

work = available

work = 3, 3, 2

finish = false.

1. for i = 0

need0 = 7,4,3

finish[0] is false and need1 >work

so p0 must wait.

1. for i = 1

need1  = 1,2,2

finish[1] is false and need1 < work

so p1 goes into the sequence.

1. work = work + allocation1

work = 5,3,2

finish = true for 1,false for 0,2,3,4.

1. for i = 2

need2 = 6,0,0

finish[2] is false and need2 >work

so p2 should wait.

1. for i = 3

need3 = 0,1,1

finsh[3] = false and need3 < work

so p3 goes to safe sequence.

1. Work = work + allocation3

Work = 7,4,3

Finish = 1,3 are true and 0,2,4 are false.

1. For i = 4

Need4 = 4,3,1

Finish[4] = false and need4 < work

So p4 goes to safe sequence.

1. Work = work + allocation4

Work = 7,4,5

Finish = 1,3,4 are true and 0,2 are false

1. For i = 0

Need0 = 7,4,3

Finish[0] is false and need < work

So p0 goes to safe sequence.

1. Work = work + allocation0

Work = 7,4,5

Finish =0,1,3,4

1. For i = 2

Need2 = 6,0,0

Finish[2] is false and need2 < work

So p2 goes to safe sequences

1. Work = work + allocation3

Work = 10,5,7

Finish all process works 0,1,2,3,4. Are true.

1. Finsh[i] = true for 0 ≤ i ≤ n

Hence the system goes to safe state.

The safe sequence is p1,p3,p4,p0,p2.

**GitHub Link:** https://github.com/Paavankumar2001/system-safe-state-detection