OPERATING SYSTEM ASSIGNMENT

SUBMITTED TO- MS. SAMREEN FAYAZ

SUBMITTED BY- VEEKENDRA PATEL

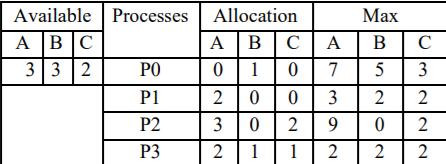
EMAIL ID- patelveekendra@gmail.com

Roll no. – 66

Reg no.-11814266

Section- K18FRG2

Ques19: 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?



C:\Users\hp\Pictures\Screenshots\Screenshot (13).png

**Description-** By definition, all the resources within a category are equivalent , and a request of this category can be equally satisfied by any one of the resources in that category.

The Banker’s algorithm gets its name because it is a method that bankers could use to assure that when they lend out resources they will still be able to satisfy all their clients. (A banker won’t loan out a little money to start building a house unless they are assured that they will later be able to loan out the rest of the money to finish the house.)

This algorithm relies on several key data structures:

* Available[m] indicates the maximum demand of each process of each resource.
* Allocation[n] [m] indicates the number of each resource category allocated to each process.
* Need [n] [m] indicate the remaining resources needed of each type for each process.
* (Need=Max-Allocation).

**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 ]

**Algorithm-**

1. Let Request [n][m] indicate the number of resources of each type currently requested by processes. If Request[i]>Need[i] for any process I, raise an error condition.
2. If Request[i]>Available for any process I, then that process must wait for resources to become available. Otherwise the process can continue to step 3.
3. Check to see if the request can be granted safely, by pretending it has been granted and then seeing if the resulting state is safe. If so, grant the request, and if not, then the process must until its request be granted safely. The procedure for granting a request(or pretending to for testing purposes) is:
   * Available=Available - Request
   * Allocation= Allocation + Request
   * Need= Need – Request

**Description (purpose of use):-**

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 ..

**Complexity of algorithm** – r\*(P\*P); where r = number of resources

**CODE SNIPPET-**

#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

int alloc[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[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 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++)

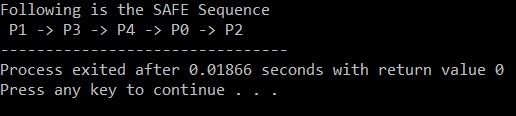
printf(" P%d ->", ans[i]);

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

return (0);

}

**OUTPUT-**



**Github link:**