**CSE316:**

**OPERATING SYSTEM**

**Prepared by : Saurav Kumar**

**Section : K18XC**

**Roll no : A31**

**U Id. : 11812088**

**Email :** [**sauravgupta.atl@gmail.com**](mailto:sauravgupta.atl@gmail.com)

**Github link :** [**https://github.com/sauravv17/cse316**](https://github.com/sauravv17/cse316)

**1.Code:**

**#include <stdio.h>**

**#include <conio.h>**

**int main()**

**{**

**int Max[10][10], need[10][10], alloc[10][10], avail[10], completed[10], safeSequence[10];**

**int p, r, i, j, process, count;**

**count = 0;**

**printf("Enter the no of processes : ");**

**scanf("%d", &p);**

**for(i = 0; i< p; i++)// complexity for this FOR LOOP is O(n)**

**completed[i] = 0;**

**printf("\n\nEnter the no of resources : ");**

**scanf("%d", &r);**

**printf("\n\nEnter the Max Matrix for each process : ");**

**for(i = 0; i < p; i++)// complexity for this FOR LOOP is O(n)**

**{**

**printf("\nFor process %d : ", i + 1);**

**for(j = 0; j < r; j++)**

**scanf("%d", &Max[i][j]);**

**}**

**printf("\n\nEnter the allocation for each process : ");**

**for(i = 0; i < p; i++)// complexity for this FOR LOOP is O(n)**

**{**

**printf("\nFor process %d : ",i + 1);**

**for(j = 0; j < r; j++)**

**scanf("%d", &alloc[i][j]);**

**}**

**printf("\n\nEnter the Available Resources : ");**

**for(i = 0; i < r; i++)// complexity for this FOR LOOP is O(n)**

**scanf("%d", &avail[i]);**

**for(i = 0; i < p; i++)// complexity for this FOR LOOP is O(n2)**

**for(j = 0; j < r; j++)**

**need[i][j] = Max[i][j] - alloc[i][j];**

**do**

**{**

**printf("\n Max matrix:\tAllocation matrix:\n");**

**for(i = 0; i < p; i++)// complexity for this FOR LOOP is O(n2)**

**{**

**for( j = 0; j < r; j++)**

**printf("%d ", Max[i][j]);**

**printf("\t\t");**

**for( j = 0; j < r; j++)**

**printf("%d ", alloc[i][j]);**

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

**}**

**process = -1;**

**for(i = 0; i < p; i++)// complexity for this FOR LOOP is O(n2)**

**{**

**if(completed[i] == 0)//if not completed**

**{**

**process = i ;**

**for(j = 0; j < r; j++)**

**{**

**if(avail[j] < need[i][j])**

**{**

**process = -1;**

**break;**

**}**

**}**

**}**

**if(process != -1)**

**break;**

**}**

**if(process != -1)**

**{**

**printf("\nProcess %d runs to completion!", process + 1);**

**safeSequence[count] = process + 1;**

**count++;**

**for(j = 0; j < r; j++)// complexity for this FOR LOOP is O(n)**

**{**

**avail[j] += alloc[process][j];**

**alloc[process][j] = 0;**

**Max[process][j] = 0;**

**completed[process] = 1;**

**}**

**}**

**}while(count != p && process != -1);// complexity for this FOR LOOP is O(n2)**

**if(count == p)**

**{**

**printf("\nThe system is in a safe state!!\n");**

**printf("Safe Sequence : < ");**

**for( i = 0; i < p; i++)**

**printf("%d ", safeSequence[i]);**

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

**}**

**else**

**printf("\nThe system is in an unsafe state!!");**

**getch();**

**}**

**2.Description:**

This question I have solved using the Banker’s Algorithm.The Banker’s algorithm is a resource allocation and deadlock avoidance algorithm developed by Edsger Dijkstra.

**Banker’s Algorithm working principle:** It tests for safety by simulating the allocation of predetermined maximum possible amounts of all resources, and then makes a “s-state” check to test for possible deadlock conditions for all other pending activities, before deciding whether allocation should be allowed to continue.

The time complexity of algorithm is r\* (P\*P) where P is the number of active processes and r is the number of resources

**3.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– Requesti

**4.Description(Purpose of use):**

|  |  |  |
| --- | --- | --- |
| **Process** | **Allocation** | **Max** |
|  | **X Y Z** | **X Y Z** |
| **P0** | **0 0 1** | **8 4 3** |
| **P1** | **3 2 0** | **6 2 0** |
| **P2** | **2 1 1** | **3 3 3** |

This is the current safe state.

|  |  |  |
| --- | --- | --- |
|  | AVAILABLE | X=3, Y=2, Z=2 |
|  |  |  |
|  | MAX | ALLOCATION |
|  | X Y Z | X Y Z |
| P0 | 8 4 3 | 0 0 1 |
| P1 | 6 2 0 | 3 2 0 |
| P2 | 3 3 3 | 2 1 1 |

Now, if the request REQ1 is permitted, the state would become :

|  |  |  |  |
| --- | --- | --- | --- |
|  | AVAILABLE | X=3, Y=2, Z=0 |  |
|  |  |  |  |
|  | MAX | ALLOCATION | NEED |
|  | X Y Z | X Y Z | X Y Z |
| P0 | 8 4 3 | 0 0 3 | 8 4 0 |
| P1 | 6 2 0 | 3 2 0 | 3 0 0 |
| P2 | 3 3 3 | 2 1 1 | 1 2 2 |

Now, with the current availability, we can service the need of P1. The state would become :

|  |  |  |  |
| --- | --- | --- | --- |
|  | AVAILABLE | X=6, Y=4, Z=0 |  |
|  |  |  |  |
|  | MAX | ALLOCATION | NEED |
|  | X Y Z | X Y Z | X Y Z |
| P0 | 8 4 3 | 0 0 3 | 8 4 0 |
| P1 | 6 2 0 | 3 2 0 | 0 0 0 |
| P2 | 3 3 3 | 2 1 1 | 1 2 2 |

With the resulting availability, it would not be possible to service the need of either P0 or P2, owing to lack of Z resource.

Therefore, the system would be in a deadlock.

⇒ We cannot permit REQ1.

Now, at the given safe state, if we accept REQ2 :

|  |  |  |  |
| --- | --- | --- | --- |
|  | AVAILABLE | X=1, Y=2, Z=2 |  |
|  |  |  |  |
|  | MAX | ALLOCATION | NEED |
|  | X Y Z | X Y Z | X Y Z |
| P0 | 8 4 3 | 0 0 1 | 8 4 2 |
| P1 | 6 2 0 | 5 2 0 | 1 0 0 |
| P2 | 3 3 3 | 2 1 1 | 1 2 2 |

With this availability, we service P1 (P2 can also be serviced). So, the state is :

|  |  |  |  |
| --- | --- | --- | --- |
|  | AVAILABLE | X=6, Y=4, Z=2 |  |
|  |  |  |  |
|  | MAX | ALLOCATION | NEED |
|  | X Y Z | X Y Z | X Y Z |
| P0 | 8 4 3 | 0 0 1 | 8 4 2 |
| P1 | 6 2 0 | 5 2 0 | 0 0 0 |
| P2 | 3 3 3 | 2 1 1 | 1 2 2 |

With the current availability, we service P2. The state becomes :

|  |  |  |  |
| --- | --- | --- | --- |
|  | AVAILABLE | X=8, Y=5, Z=3 |  |
|  |  |  |  |
|  | MAX | ALLOCATION | NEED |
|  | X Y Z | X Y Z | X Y Z |
| P0 | 8 4 3 | 0 0 1 | 8 4 2 |
| P1 | 6 2 0 | 5 2 0 | 0 0 0 |
| P2 | 3 3 3 | 2 1 1 | 0 0 0 |

Finally, we service P0. The state now becomes :

|  |  |  |  |
| --- | --- | --- | --- |
|  | AVAILABLE | X=8, Y=5, Z=4 |  |
|  |  |  |  |
|  | MAX | ALLOCATION | NEED |
|  | X Y Z | X Y Z | X Y Z |
| P0 | 8 4 3 | 0 0 1 | 0 0 0 |
| P1 | 6 2 0 | 5 2 0 | 0 0 0 |
| P2 | 3 3 3 | 2 1 1 | 0 0 0 |

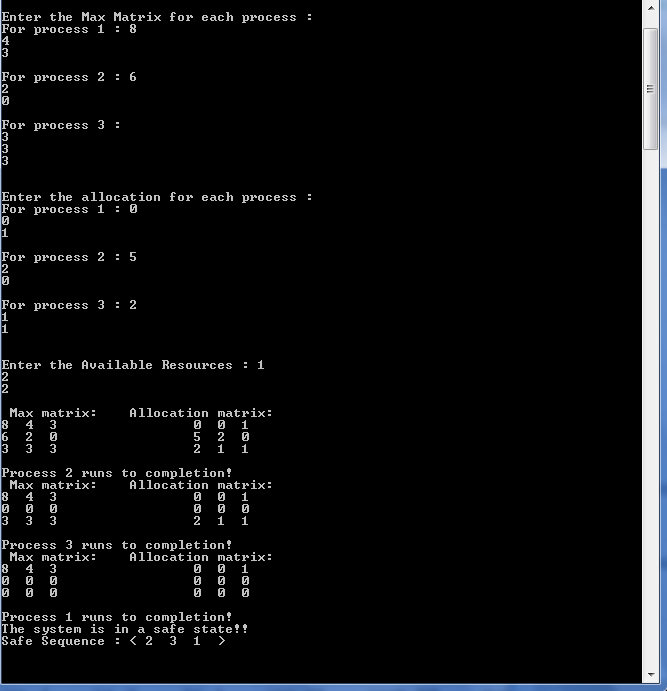
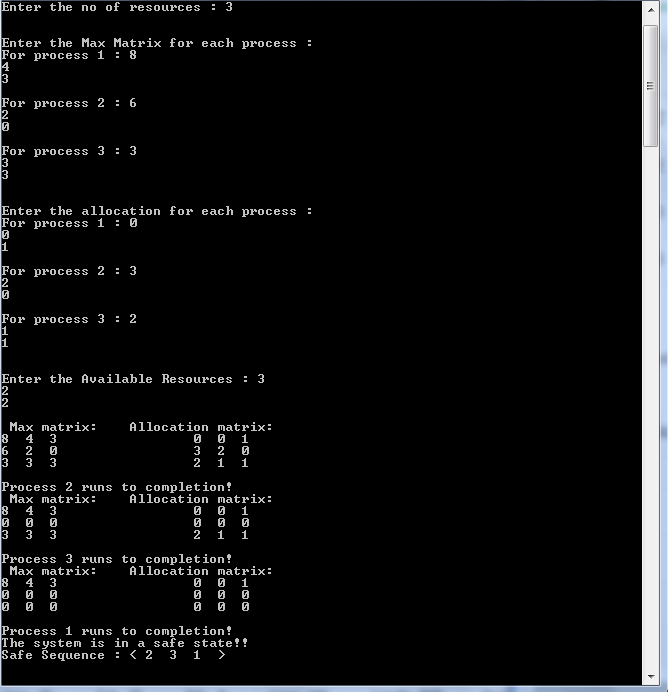
The state so obtained is a safe state. ⇒ REQ2 can be permitted.

So, only REQ2 can be permitted.

**5.Test cases:**

1. Request1: P0 requests 0 instances of A and 0 instances of B and 2 instances of C.
2. Request2: P1 requests for 2 instances of A, 0 instances of B and 0 instances of C.

**6.Code Snippet:**

**7.Github Link:**

[**https://github.com/sauravv17/cse316**](https://github.com/sauravv17/cse316)