

Operating Systems

Student Name: **Jovin Kumar S**

Student ID: **11801466**

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

Roll No:**36**

**Code:**

#include<conio.h>

#include<stdio.h>

int counter = 0, i, j, exec, resources, processes, k = 1;

int allocation[5] = {0, 0, 0, 0, 0};

int maxres[5], running[5], safe = 0;

int current[5][5], maximum\_claim[5][5], available[5];

int main()

{

printf("\nEnter number of processes: ");

scanf("%d", &processes);

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

{

running[i] = 1;

counter++;

}

printf("\nEnter number of resources: ");

scanf("%d", &resources);

printf("\nEnter Claim Vector:");

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

{

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

}

printf("\nEnter Allocated Resource Table:\n");

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

{

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

{

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

}

}

printf("\nEnter Maximum Claim Table:\n");

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

{

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

{

scanf("%d", &maximum\_claim[i][j]);

}

}

printf("\nThe Claim Vector is: ");

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

{

printf("\t%d", maxres[i]);

}

printf("\nThe Allocated Resource Table:\n");

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

{

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

{

printf("\t%d", current[i][j]);

}

printf("\n");

}

printf("\nThe Maximum Claim Table:\n");

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

{

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

{

printf("\t%d", maximum\_claim[i][j]);

}

printf("\n");

}

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

{

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

{

allocation[j] += current[i][j];

}

}

printf("\nAllocated resources:");

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

{

printf("\t%d", allocation[i]);

}

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

{

available[i] = maxres[i] - allocation[i];

}

printf("\nAvailable resources:");

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

{

printf("\t%d", available[i]);

}

printf("\n");

while (counter != 0)

{

safe = 0;

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

{

if (running[i])

{

exec = 1;

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

{

if (maximum\_claim[i][j] - current[i][j] > available[j])

{

exec = 0;

break;

}

}

if (exec)

{

printf("\nProcess%d is executing\n", i + 1);

running[i] = 0;

counter--;

safe = 1;

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

{

available[j] += current[i][j];

}

break;

}

}

}

if (!safe)

{

printf("\nThe processes are in unsafe state.\n");

break;

}

else

{

printf("\nThe process is in safe state");

printf("\nAvailable vector:");

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

{

printf("\t%d", available[i]);

}

printf("\n");

}

}

return 0;

}

Example:

Suppose P0 has 0,0,1 instances , P1 is having 3,2,0 instances and P2 occupies 2,1,1 instances of A,B,C resource respectively.

Also the maximum number of instances required for P0 is 8,4,3 and for p1 is 6,2,0 and finally for P2 there are 3,3,3 instances of resources A,B,C respectively. There are 3 instances of resource A, 2 instances of resource B and 2 instances of resource C available

Explanation:

|  |  |  |
| --- | --- | --- |
|  | 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.

Hence, B is the correct choice.

2. Algoritm:

**Step-01:**

With the instances available currently, only the requirement of the process P1 can be satisfied.

So, process P1 is allocated the requested resources.

It completes its execution and then free up the instances of resources held by it.

**Step-02:**

With the instances available currently, only the requirement of the process P2 can be satisfied.

So, process P2 is allocated the requested resources.

It completes its execution and then free up the instances of resources held by it.

**Step-03:**

With the instances available currently, the requirement of the process P0 can be satisfied.

So, process P0 is allocated the requested resources.

It completes its execution and then free up the instances of resources held by it.

 Thus,

There exists a safe sequence P1, P2, P0 in which all the processes can be executed.

So, the system is in a safe state.

Thus, REQ2 can be permitted.

And deadlock.execution file: