**Student Name: -** P. Amar Krishna

**Student ID:-**11706635 ; **ROLL-NUMBER :-**31 ; **Question number:-** 10

**Email Address:-** [Amarkrishna230@gmail.com](mailto:Amarkrishna230@gmail.com)

**Github Link:-** <https://github.com/Amarkrishna230>

**Code:-**

#include <stdio.h>

int main(){

int b = 0;

int count = 0, m, n, process, temp, resource;

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

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

int maximum\_resources[5], running[5], safe\_state = 0;

printf("\nEnter The Total Number Of Processes:\t");

scanf("%d", &process);

for(m = 0; m < process; m++) {

running[m] = 1;

count++;

}

printf("\nEnter The Total Number Of Resources To Allocate:\t");

scanf("%d", &resource);

printf("\nEnter The Claim Vector:\t");

for(m = 0; m < resource; m++) {

scanf("%d", &maximum\_resources[m]);

}

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

for(m = 0; m < process; m++) {

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

{

scanf("%d", &current[m][n]);

}

}

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

for(m = 0; m < process; m++) {

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

{

scanf("%d", &maximum\_claim[m][n]);

}

}

printf("\nThe Claim Vector \n");

for(m = 0; m < resource; m++) {

printf("\t%d ", maximum\_resources[m]);

}

printf("\n The Allocated Resource Table\n");

for(m = 0; m < process; m++) {

for(n = 0; n < resource; n++) {

printf("\t%d", current[m][n]);

}

printf("\n");

}

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

for(m = 0; m < process; m++) {

for(n = 0; n < resource; n++) {

printf("\t%d", maximum\_claim[m][n]);

}

printf("\n");

}

for(m = 0; m < process; m++) {

for(n = 0; n < resource; n++) {

allocation\_table[n] = allocation\_table[n] + current[m][n];

}

}

printf("\nAllocated Resources \n");

for(m = 0; m < resource; m++) {

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

}

for(m = 0; m < resource; m++) {

available[m] = maximum\_resources[m] - allocation\_table[m];

}

printf("\nAvailable Resources:");

for(m = 0; m < resource; m++) {

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

}

printf("\n");

while(count != 0) {

safe\_state = 0;

for(m = 0; m < process; m++) {

if(running[m]) {

temp = 1;

for(n = 0; n < resource; n++) {

if(maximum\_claim[m][n] - current[m][n] > available[n]) {

temp = 0;

break;

}

}

if(temp) {

printf("\nProcess %d Is In Execution \n", m + 1);

running[m] = 0;

count--;

safe\_state = 1;

for(n = 0; n < resource; n++) {

available[n] = available[n] + current[m][n];

}

break;

}

}

}

if(!safe\_state) {

printf("\nThe Processes Are In An Unsafe State \n");

break;

}

else {

printf("\nThe Process Is In A Safe State \n");

printf("\nAvailable Vector\n");

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

{

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

}

printf("\n");

}

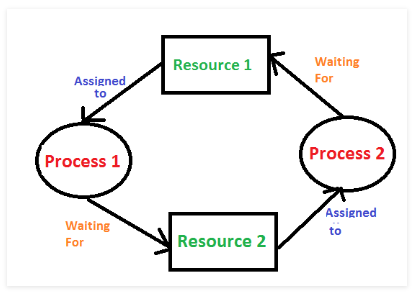
}

return 0;

}

**DESCRIPTION OF PROBLEM:-**

**Deadlock**is a situation where a set of processes are blocked because each process is holding a resource and waiting for another resource acquired by some other process.  
Consider an example when two trains are coming toward each other on same track and there is only one track, none of the trains can move once they are in front of each other. Similar situation occurs in operating systems when there are two or more processes hold some resources and wait for resources held by other(s). For example, in the below diagram, Process 1 is holding Resource 1 and waiting for resource 2 which is acquired by process 2, and process 2 is waiting for resource 1.



**Deadlock can arise if following four conditions hold simultaneously (Necessary Conditions)**  
**Mutual Exclusion:** One or more than one resource are non-sharable (Only one process can use at a time)  
**Hold and Wait:** A process is holding at least one resource and waiting for resources.  
**No Preemption*:*** A resource cannot be taken from a process unless the process releases the resource.  
**Circular Wait:** A set of processes are waiting for each other in circular form.

**Methods for handling deadlock**  
There are ways to handle deadlock one of them is :-  
 **Deadlock prevention or avoidance:** The idea is to not let the system into deadlock state. Avoidance is kind of futuristic in nature. By using strategy of “Avoidance”, we have to make an assumption. We need to ensure that all information about resources which process WILL need are known to us prior to execution of the process. We use Banker’s algorithm (Which is in-turn a gift from Dijkstra) in order to avoid deadlock.

**ALGORITHM USED FOR THIS PROBLEM:-**

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.  
Banker’s algorithm is named so because it is used in banking system to check whether loan can be sanctioned to a person or not. Suppose there are n number of account holders in a bank and the total sum of their money is S. If a person applies for a loan then the bank first subtracts the loan amount from the total money that bank has and if the remaining amount is greater than S then only the loan is sanctioned. It is done because if all the account holders comes to withdraw their money then the bank can easily do it.

In other words, the bank would never allocate its money in such a way that it can no longer satisfy the needs of all its customers. The bank would try to be in safe state always.

**UNDERSTANDING THE TERMS:-** 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 ]

**ALGORITHM OF SAFE STATE:-**

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…

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

**ALGORITHM OF REQUEST STATE:-**

Let Request be the request array for process Pi. Request[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 Request <= Need  
Goto step (2); otherwise, raise an error condition, since the process has exceeded its maximum claim.

2) If Request<= 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 – Request  
Allocation = Allocation + Request  
Need = Need– Request

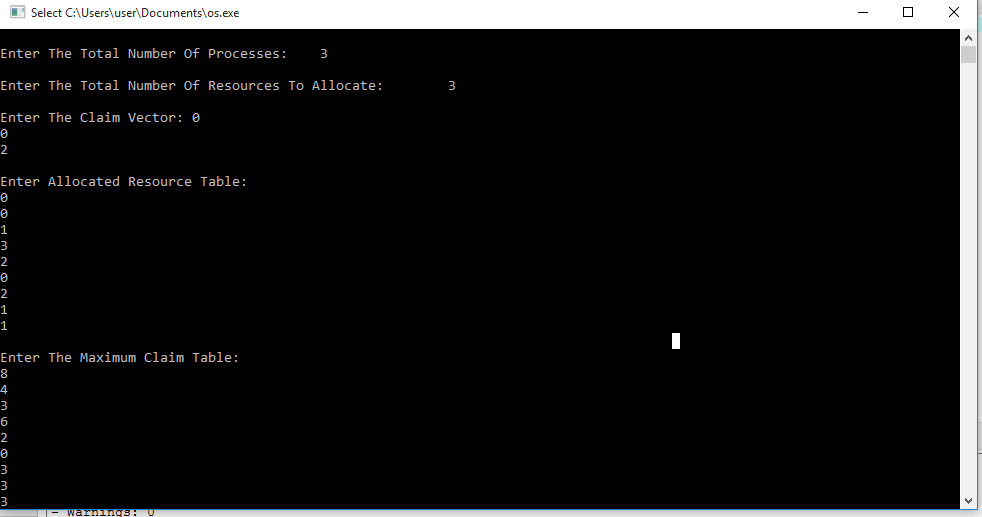
**Time complexity of the algorithm:-**

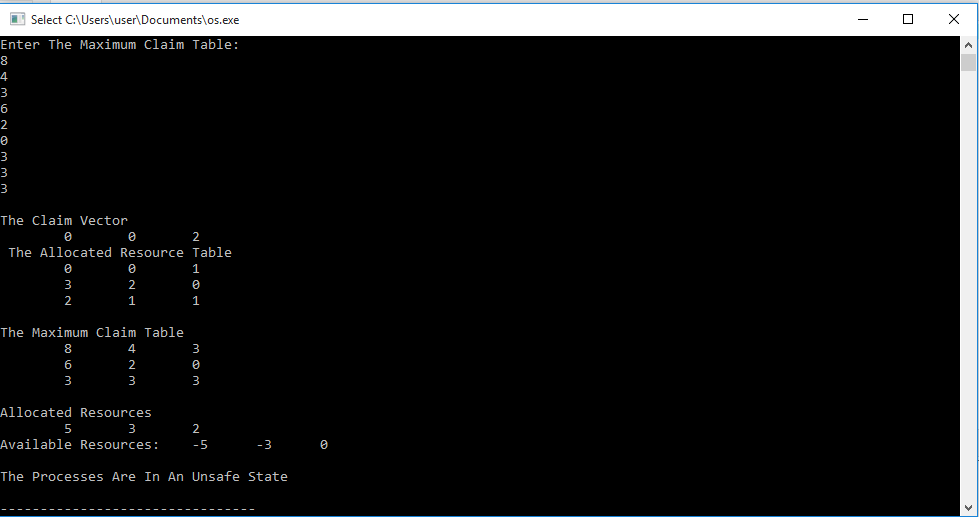
The time complexity of banker's algorithm is r\* (P\*P) where P is the number of active processes and r is the number of resources.

**BOUNDARY CONDITIONS OF THE CODE:-**

1. The number of processes cannot be or less than zero
2. Atleast there should be one resource to allocate to the resources
3. The request vector or the claim vector should be equal in length with the number of resources that is each process should make a request
4. If request is exceeded the (maximum-current) it should break

**OUTPUT OF THE SOLUTION WITH GIVEN TEST CASE CONSTRAINTS:**

****

****