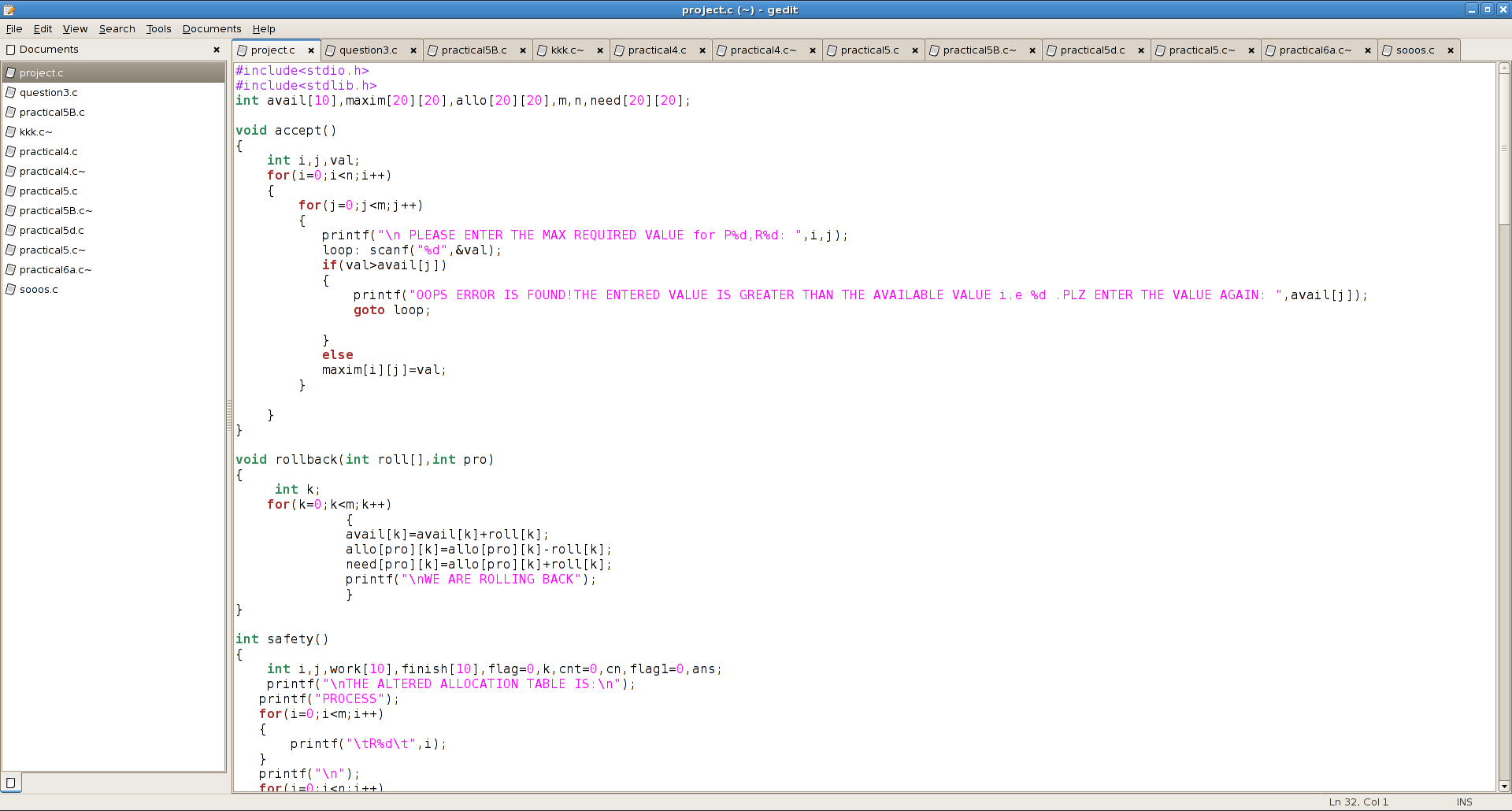
**Name : Cabinet shah**

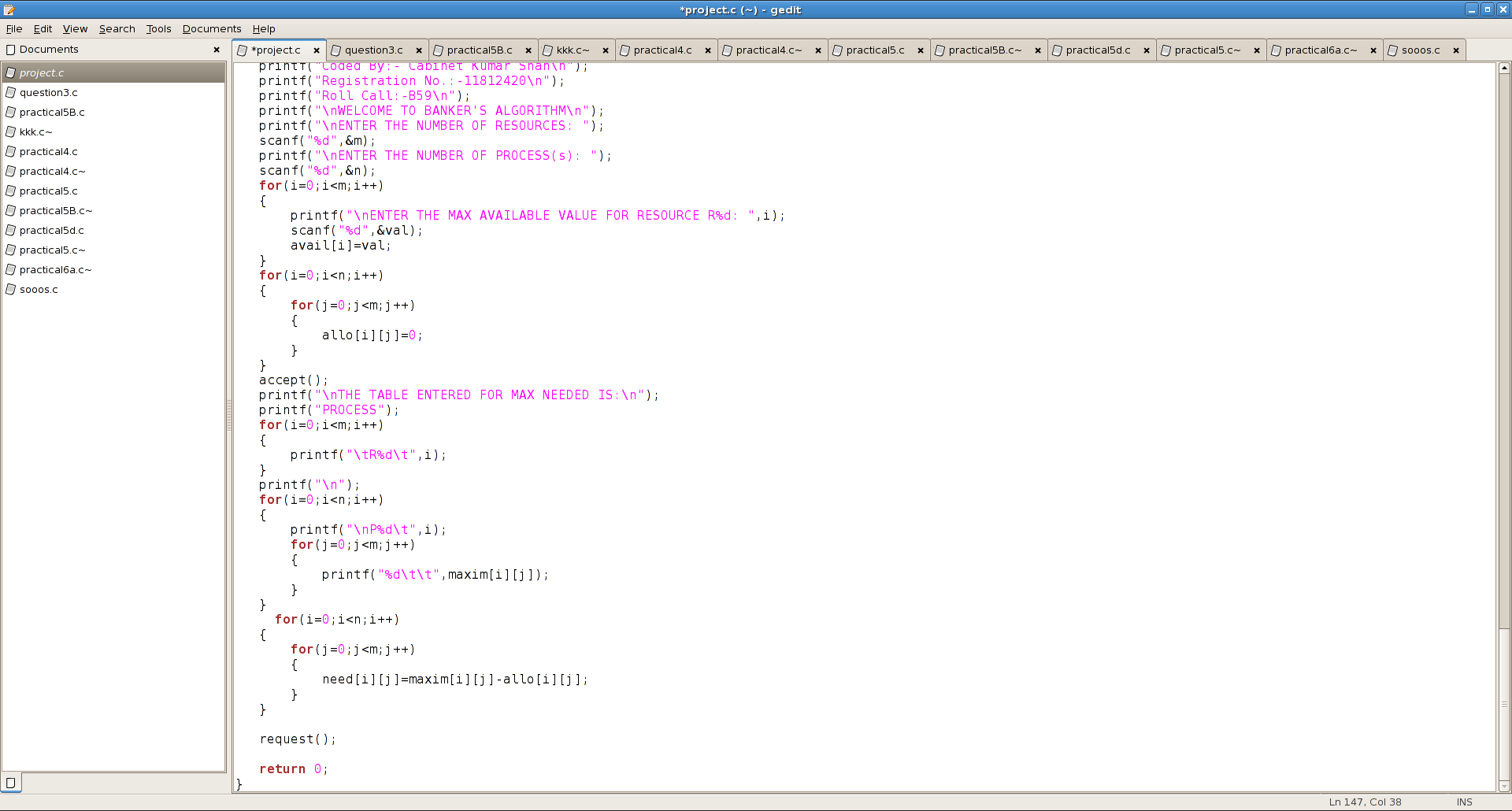
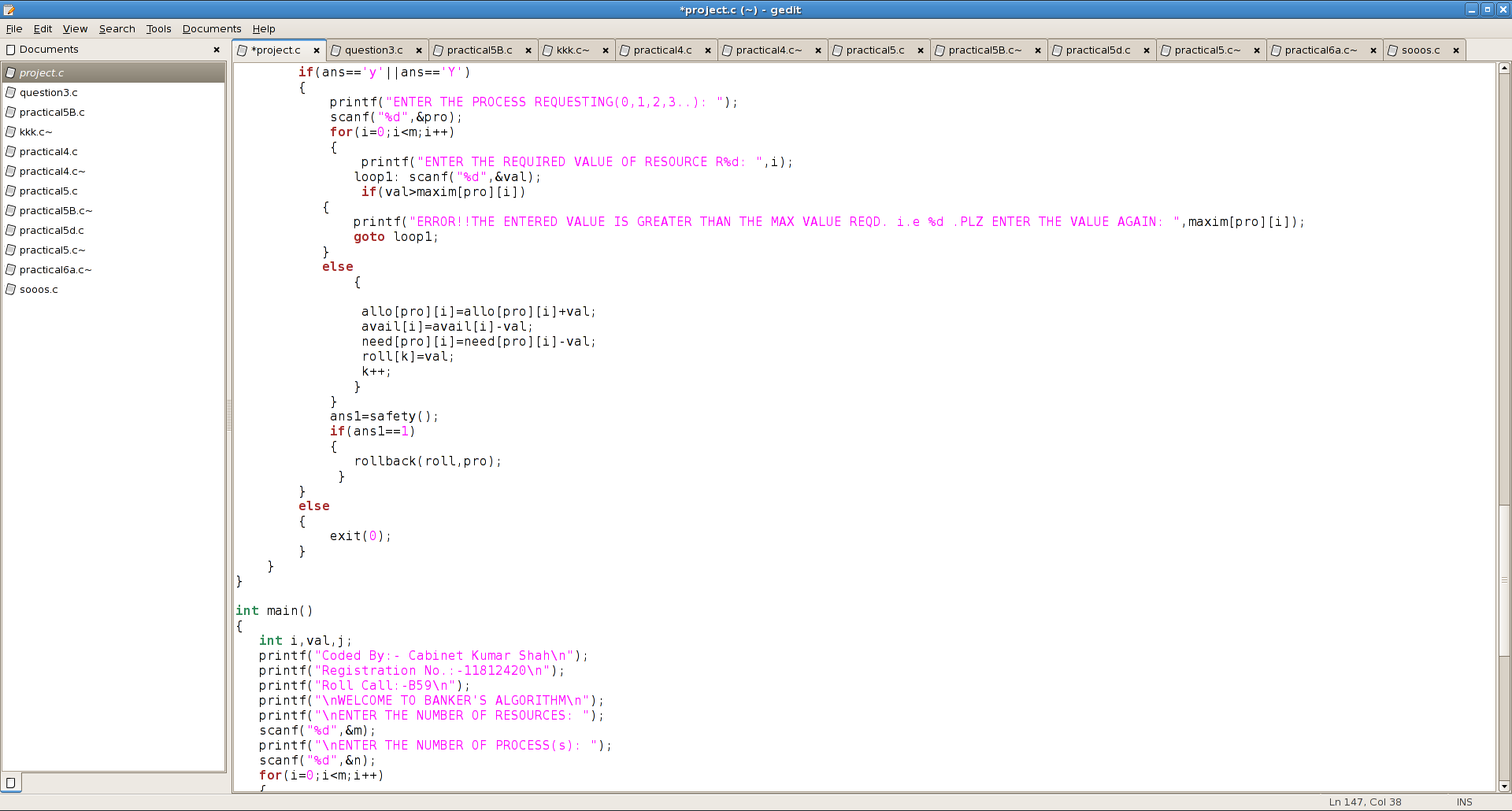
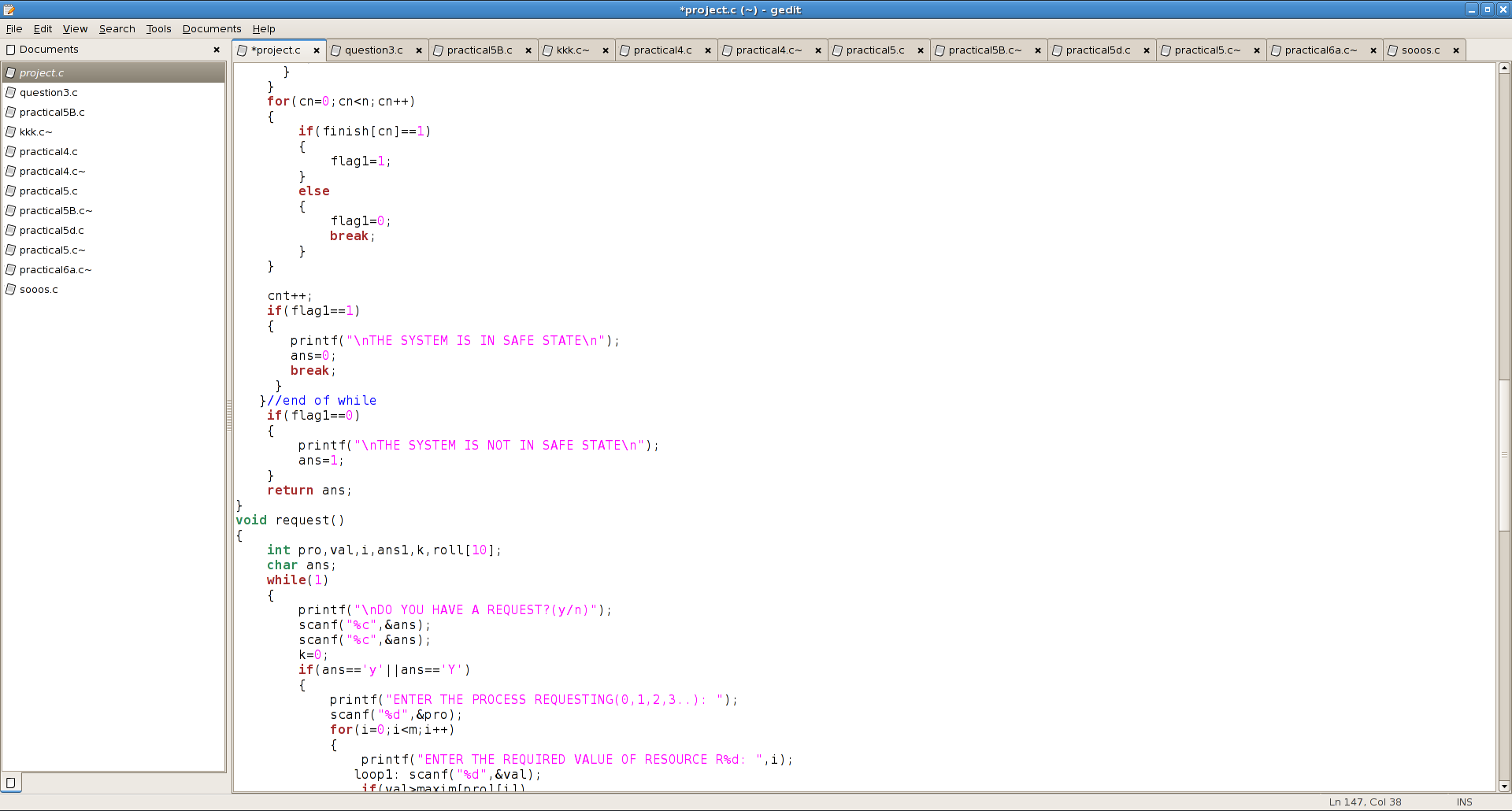
**Registration No. : 11812420**

**Email Address :** [**cabinetshah2710@gmail.com**](mailto:cabinetshah2710@gmail.com)

**GitHub link :** [**https://github.com/cabinetshah2710**](https://github.com/cabinetshah2710)

**Project Code :**

****

****

**Introduction (OS concept)**

Since the project topic allotted is based upon Banker’s Algorithm. In this project I have tried to fulfill all the constraints given in the topic. I have completed the code with my following OS concept.

Proof

Suppose N= sum of all need (i),

A= sum of all Allocation (i),

M= sum of all Max (i)

Using contradiction to prove,

Assume, this system is not deadlock free if there exists a deadlock state, then A=m because there’s only one kind of resources and resources can be requested and released only at a time.

Firstly I will be proving 2nd condition given in the question.

--from the 2nd condition,

N+A = M < m+n -> so we get N+m < m+n

Thus, we get N < n

Hence It shows that at least one process i that need (i)=0.

--from the 1st condition,

Pi can release at least 1 resource. So there are (n-1) process sharing m resource now.

Finally, we can condition 1st and 2nd still hold. No process will wait permanently, so theirs is no deadlock.

**Algorithm**

**Banker’s Algorithm**

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.

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 ]

Allocationi specifies the resources currently allocated to process Pi and Needi specifies the additional resources that process Pi may still request to complete its task.

**Calculating the complexity of implemented algorithm**

#include<stdio.h>

#include<stdlib.h>

int avail[10],maxim[20][20],allo[20][20],m,n,need[20][20];

void accept()

{

int i,j,val;

for(i=0;i<n;i++) **complexity of line = O(n \* m)**

{

for(j=0;j<m;j++) **complexity of line = O(m)**

{

printf("\n PLEASE ENTER THE MAX REQUIRED VALUE for P%d,R%d: ",i,j);

loop: scanf("%d",&val);

if(val>avail[j]) **complexity of line = O(1)**

{

printf("OOPS ERROR IS FOUND!THE ENTERED VALUE IS GREATER THAN THE AVAILABLE VALUE i.e %d .PLZ ENTER THE VALUE AGAIN: ",avail[j]);

goto loop;

}

else **complexity of line = O(1)**

maxim[i][j]=val;

}

}

}

void rollback(int roll[],int pro)

{

int k;

for(k=0;k<m;k++) **complexity of line = O(m)**

{

avail[k]=avail[k]+roll[k];

allo[pro][k]=allo[pro][k]-roll[k];

need[pro][k]=allo[pro][k]+roll[k];

printf("\nWE ARE ROLLING BACK");

}

}

int safety()

{

int i,j,work[10],finish[10],flag=0,k,cnt=0,cn,flag1=0,ans;

printf("\nTHE ALTERED ALLOCATION TABLE IS:\n");

printf("PROCESS");

for(i=0;i<m;i++) **complexity of line = O(m)**

{

printf("\tR%d\t",i);

}

printf("\n");

for(i=0;i<n;i++) **complexity of line = O(n \* m)**

{

printf("\nP%d\t",i);

for(j=0;j<m;j++) **complexity of line = O(m)**

{

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

}

}

j=0;

for(i=0;i<m;i++) **complexity of line = O(m)**

{

work[j]=avail[i];

j++;

}

for(i=0;i<n;i++) **complexity of line = O(n)**

{

finish[i]=0;

}

while(cnt<=10)

{

for(i=0;i<n;i++) **complexity of line = O(n \* m)**

{

if(finish[i]==0) **complexity of line = O(m)**

{

for(j=0;j<m;j++) **complexity of line = O(m)**

{

if(need[i][j]<=work[j])

{

flag=1;

}

else **complexity of line = O(1)**

{

flag=0;

break;

}

}

if(flag==1) **complexity of line = O(1)**

{

printf("\nTHE PROCESS P%d RAN SUCCESSFULLY",i);

finish[i]=1;

for(k=0;k<m;k++) **complexity of line = O(m)**

{

work[k]=work[k]+allo[i][k];

}

}

}

}

for(cn=0;cn<n;cn++) **complexity of line = O(cn)**

{

if(finish[cn]==1) **complexity of line = O(1)**

{

flag1=1;

}

else **complexity of line = O(1)**

{

flag1=0;

break;

}

}

cnt++;

if(flag1==1) **complexity of line = O(1)**

{

printf("\nTHE SYSTEM IS IN SAFE STATE\n");

ans=0;

break;

}

}//end of while

if(flag1==0) **complexity of line = O(1)**

{

printf("\nTHE SYSTEM IS NOT IN SAFE STATE\n");

ans=1;

}

return ans;

}

void request()

{

int pro,val,i,ans1,k,roll[10];

char ans;

while(1) **complexity of line = O(m)**

{

printf("\nDO YOU HAVE A REQUEST?(y/n)");

scanf("%c",&ans);

scanf("%c",&ans);

k=0;

if(ans=='y'||ans=='Y') **complexity of line = O(m)**

{

printf("ENTER THE PROCESS REQUESTING(0,1,2,3..): ");

scanf("%d",&pro);

for(i=0;i<m;i++) **complexity of line = O(m)**

{

printf("ENTER THE REQUIRED VALUE OF RESOURCE R%d: ",i);

loop1: scanf("%d",&val);

if(val>maxim[pro][i]) **complexity of line = O(1)**

{

printf("ERROR!!THE ENTERED VALUE IS GREATER THAN THE MAX VALUE REQD. i.e %d .PLZ ENTER THE VALUE AGAIN: ",maxim[pro][i]);

goto loop1;

}

else **complexity of line = O(1)**

{

allo[pro][i]=allo[pro][i]+val;

avail[i]=avail[i]-val;

need[pro][i]=need[pro][i]-val;

roll[k]=val;

k++;

}

}

ans1=safety();

if(ans1==1) **complexity of line = O(1)**

{

rollback(roll,pro);

}

}

else **complexity of line = O(1)**

{

exit(0);

}

}

}

int main()

{

int i,val,j;

printf("Coded By:- Cabinet Kumar Shah\n");

printf("Registration No.:-11812420\n");

printf("Roll Call:-B59\n");

printf("\nWELCOME TO BANKER'S ALGORITHM\n");

printf("\nENTER THE NUMBER OF RESOURCES: ");

scanf("%d",&m);

printf("\nENTER THE NUMBER OF PROCESS(s): ");

scanf("%d",&n);

for(i=0;i<m;i++) **complexity of line = O(m)**

{

printf("\nENTER THE MAX AVAILABLE VALUE FOR RESOURCE R%d: ",i);

scanf("%d",&val);

avail[i]=val;

}

for(i=0;i<n;i++) **complexity of line = O(n \* m)**

{

for(j=0;j<m;j++) **complexity of line = O(m)**

{

allo[i][j]=0;

}

}

accept();

printf("\nTHE TABLE ENTERED FOR MAX NEEDED IS:\n");

printf("PROCESS");

for(i=0;i<m;i++) **complexity of line = O(m)**

{

printf("\tR%d\t",i);

}

printf("\n");

for(i=0;i<n;i++) **complexity of line = O(n \* m)**

{

printf("\nP%d\t",i);

for(j=0;j<m;j++) **complexity of line = O(m)**

{

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

}

}

for(i=0;i<n;i++) **complexity of line = O(n \* m)**

{

for(j=0;j<m;j++) **complexity of line = O(m)**

{

need[i][j]=maxim[i][j]-allo[i][j];

}

}

request();

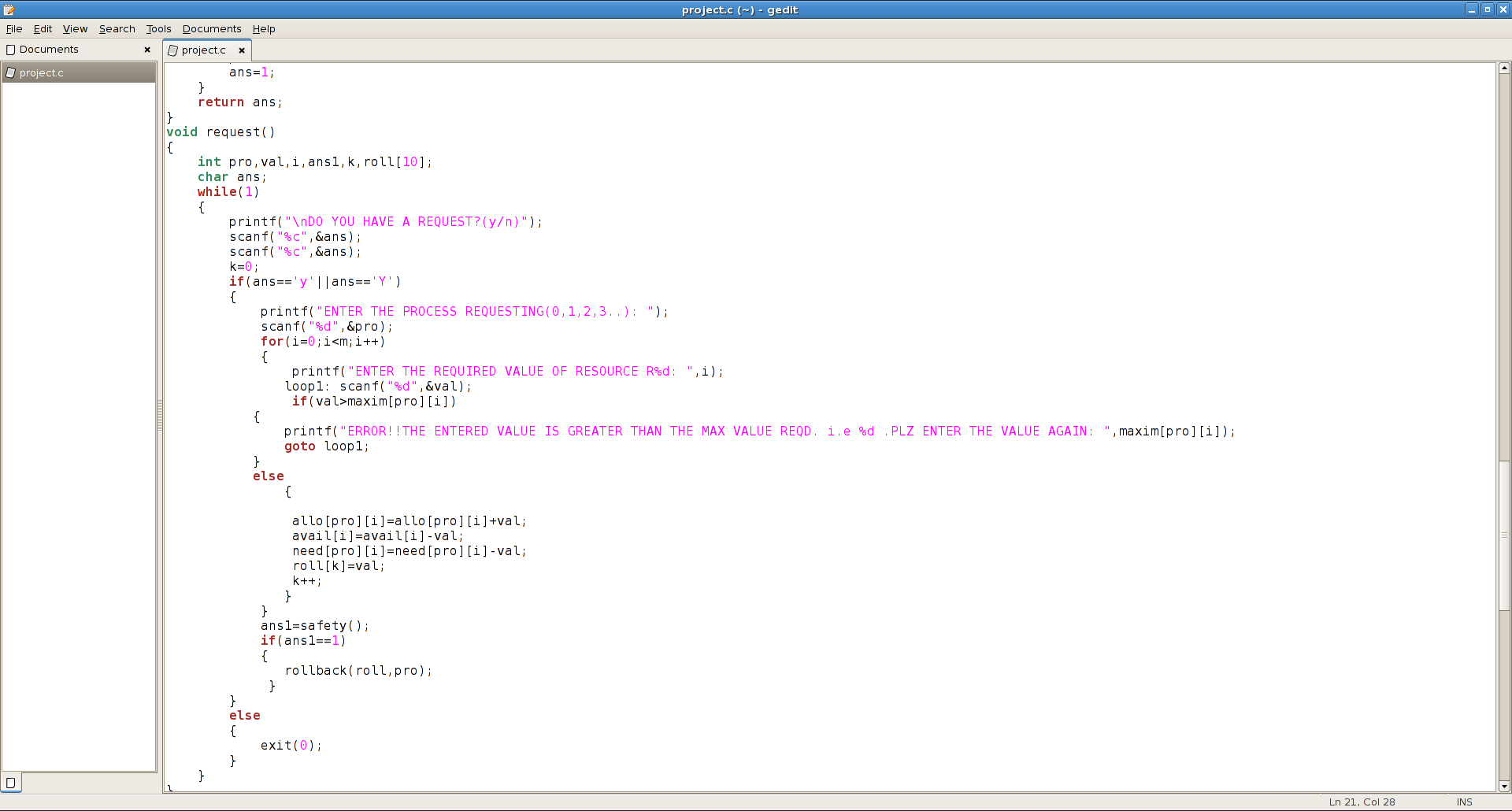
return 0;

}

**Code snippet**

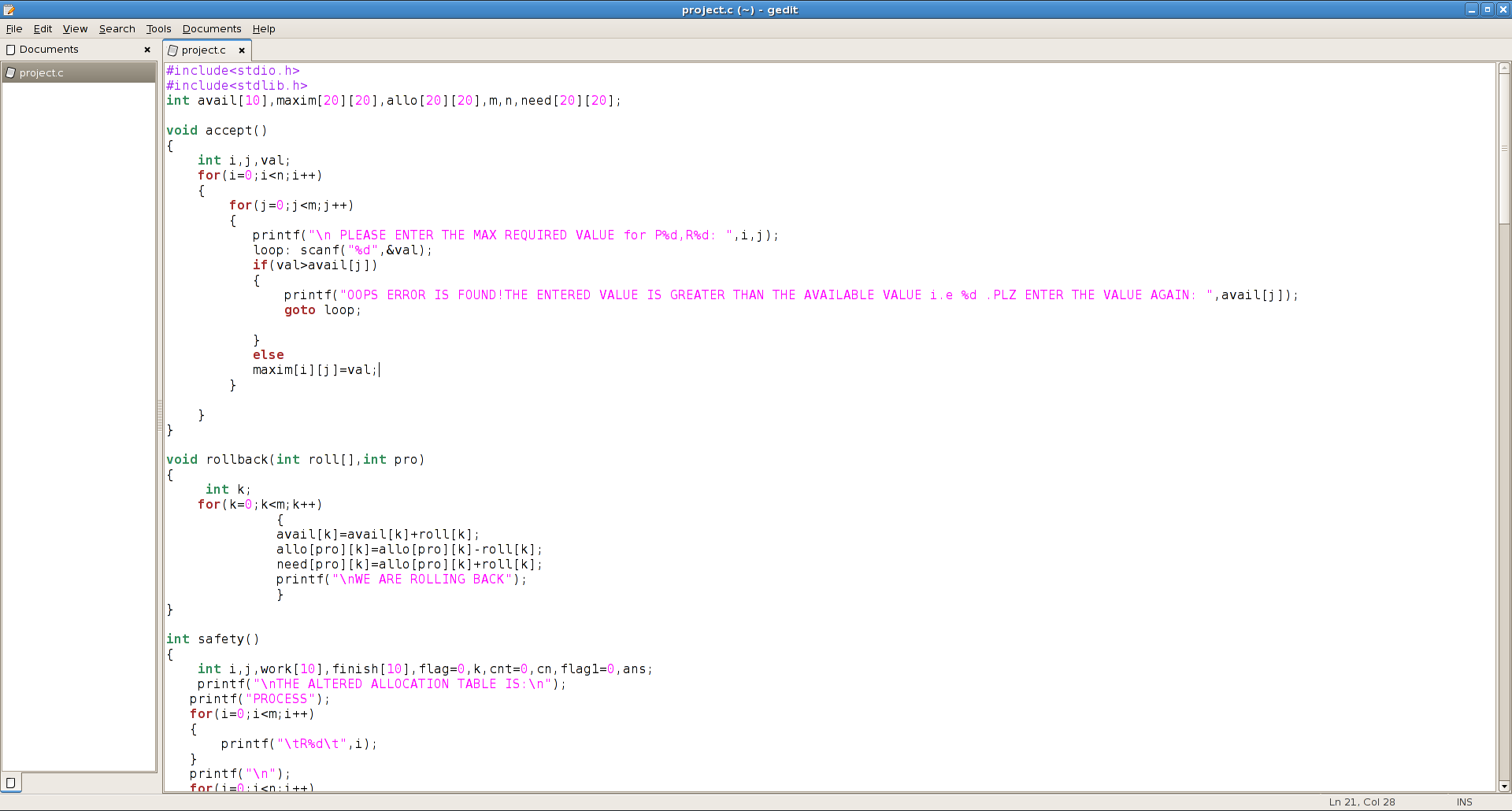
1. Maximum resource need of each process is between 1 and P.

When we exceed the value p it will generate message saying you have exceed the range.



1. Summation of all maximum needs is less than P+Q

Whenever the max need will exceed the total of p+q it will generate the error message saying that the need has exceed.

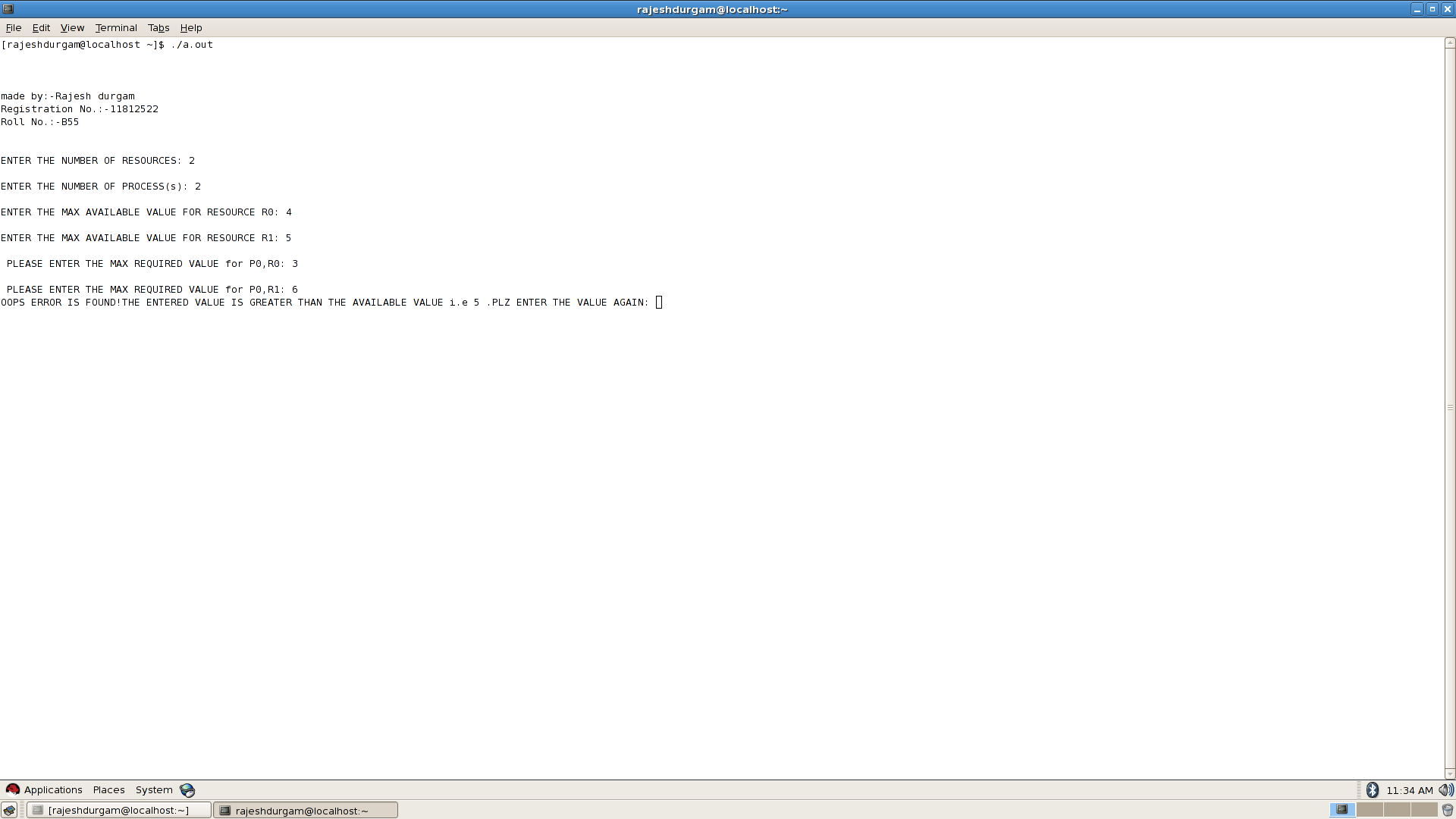


**Boundary Conditions Of The Implemented Code.**

1. **The request will only be granted under the below conditions:**
2. If the request made by the process is less than equal to max need to that process.
3. If the request made by the process is less than equal to the freely available resource in the system
4. **Inputs to Banker’s Algorithm:**
5. Max need of resources by each process.
6. Currently allocated resources by each process.
7. Max free available resources in the system
8. **Maximum resource need of each process is between 1 and the resources given for the system.**
9. **Summation of all maximum needs is less than summation of the resources and the processes.**
10. **Multiple Instances of Resource Type**
11. **Multiple instances of the resource type is a situation where a system is having multiple instances of all resources. According to the Bankers algorithm, as soon as the process gets all its required resources, then it releases its resources.**
12. **Deadlock avoidance algorithm which test all the request made by processes for resources, it checks for the safe state, if after granting request system remains in the safe state it allows the request and if there is no safe state it doesn’t allow the request made by the process.**

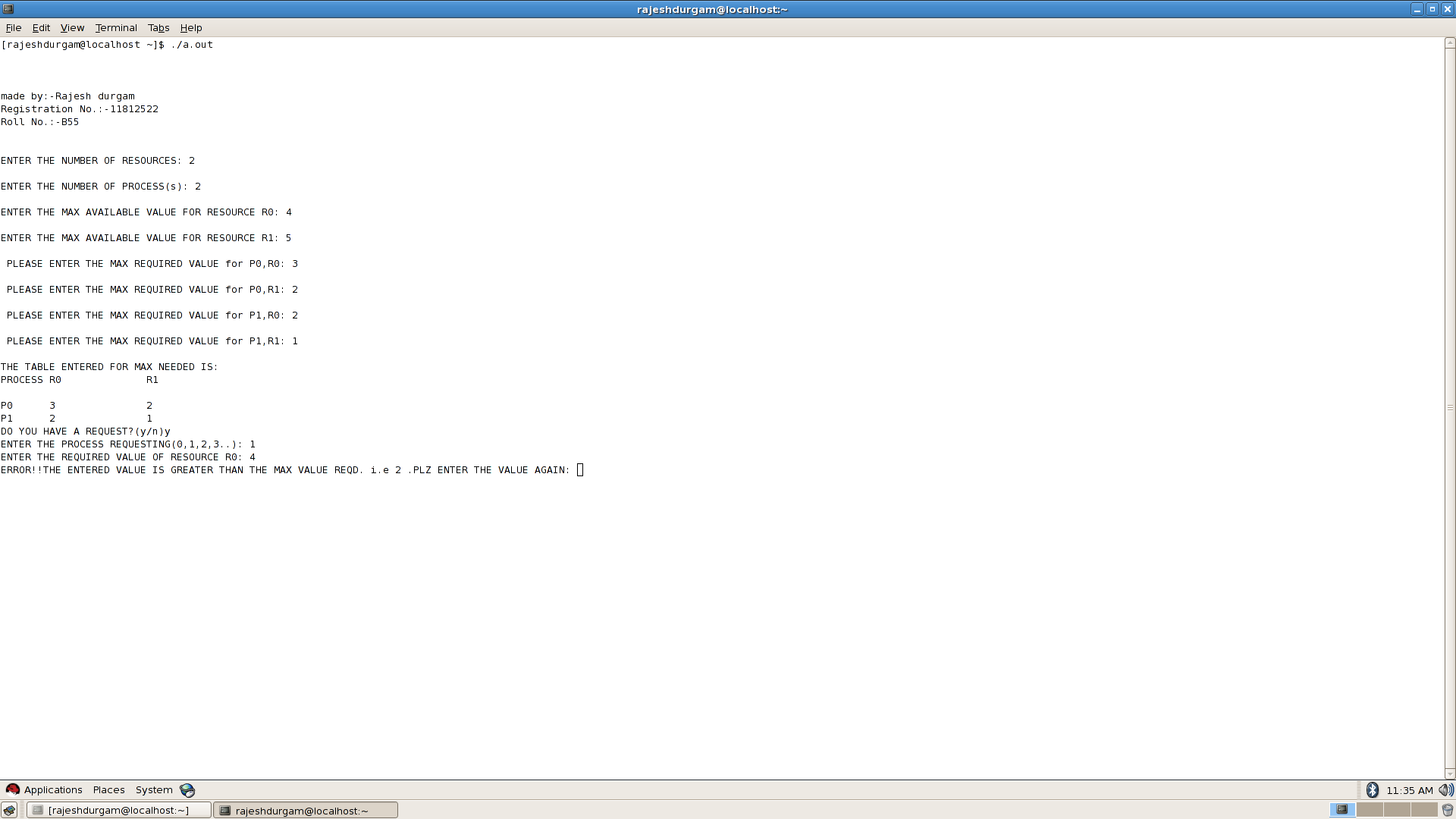
**Explain all the test cases applied on the solution of assigned problem.**

**1)**

****

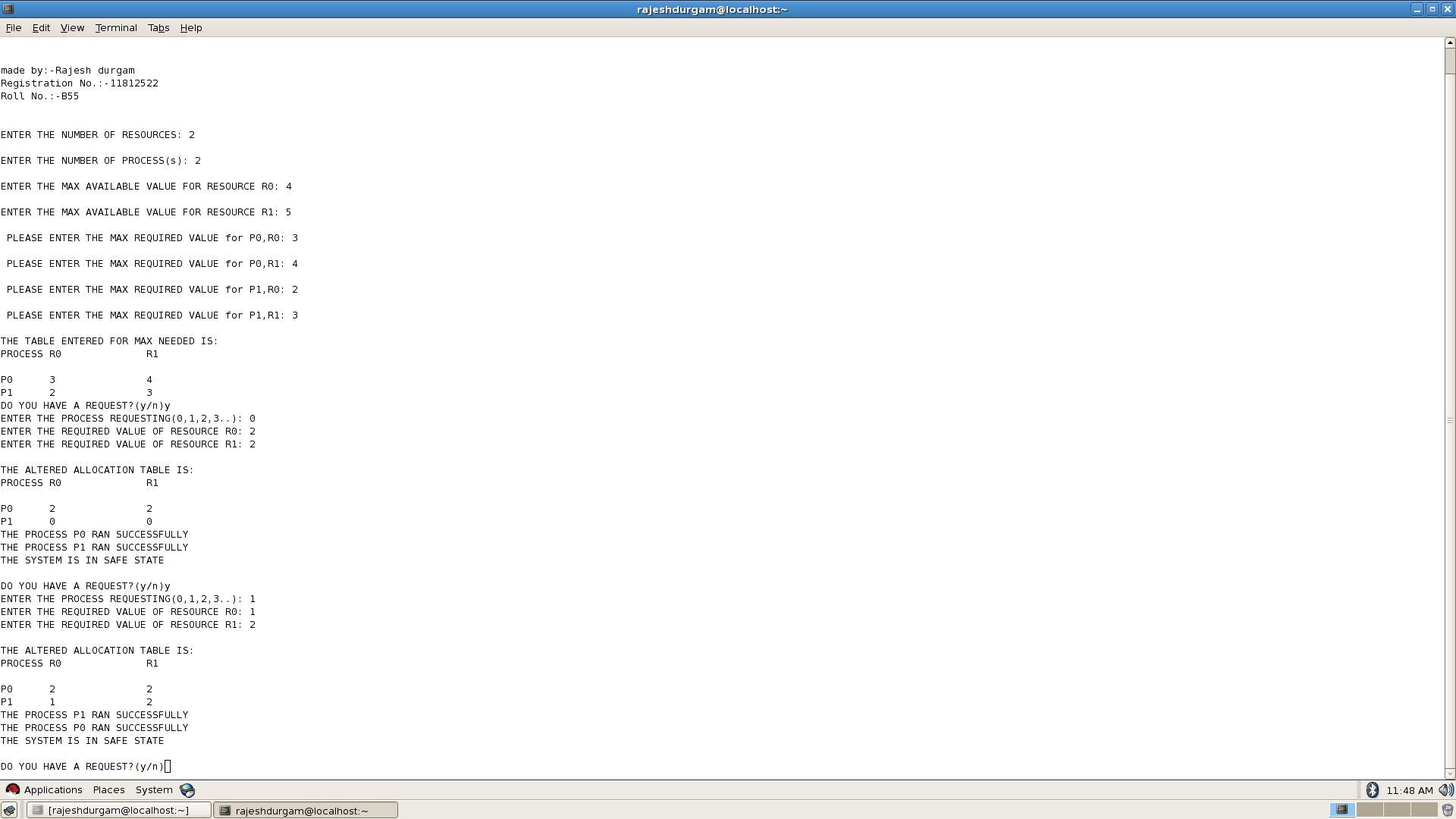
As shown in the above screen shot if the maximum required value is greater than the maximum available value. Then it shows the error. **Hence Maximum required value should be less than or equals to the Maximum available value.**

**2)**

****

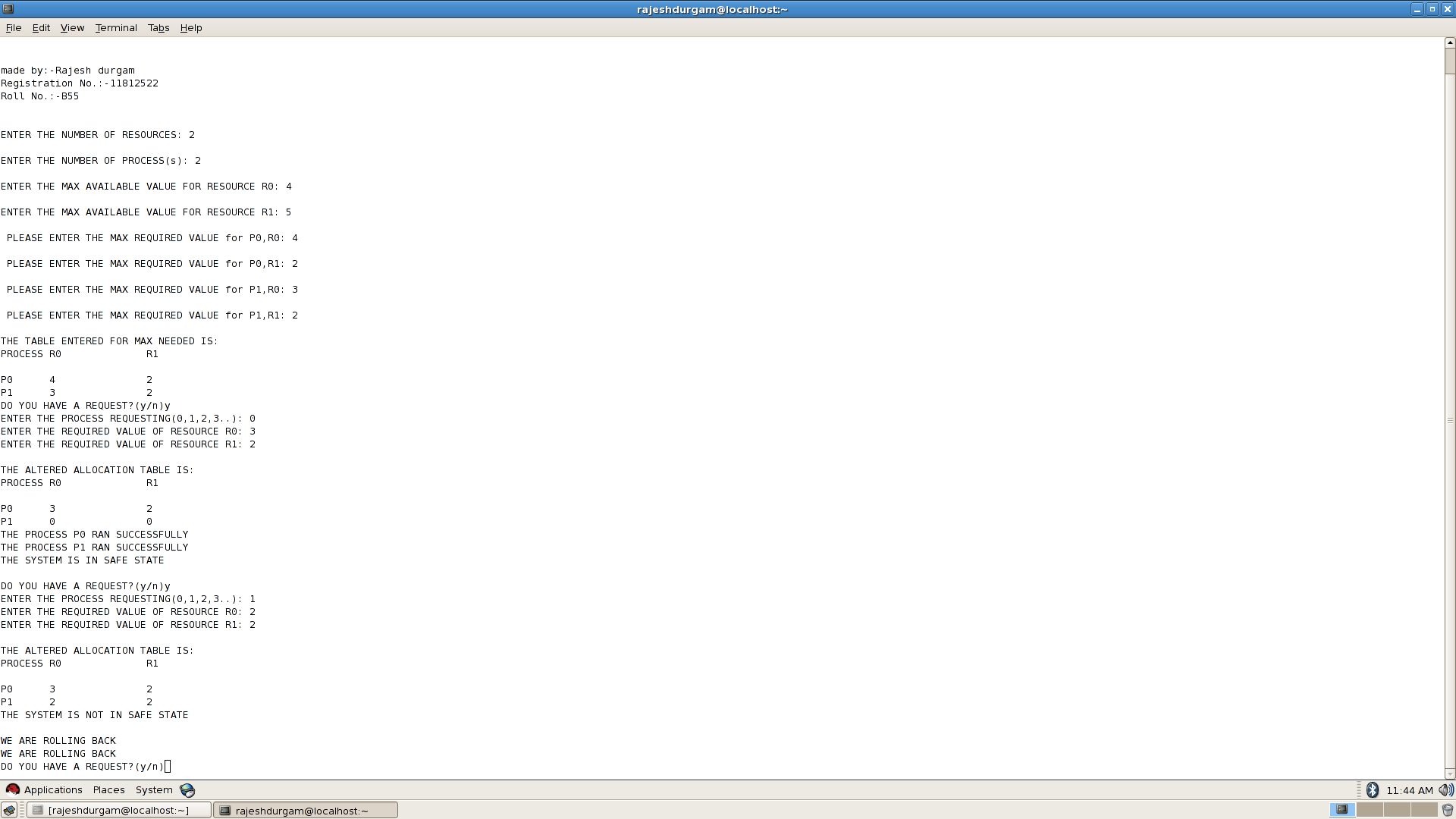
As shown in the above screen shot required value of the resource should be less than the Maximum value required.

**3)**



As shown in the above screen shot summation of the value of the resources’ in the allocation table should be less than the Maximum available value for the resources. Hence **the system is in safe condition.**

4)



As shown In the above screen shot. I f the value of the resources’ in the allocation table is greater than the Maximum available value for the resources. Hence the **system is Not in safe condition.**

**Github Repositories link:**

[**https://github.com/cabinetshah2710/Operating-system-project-on-Banker-s-Algorithm/tree/master**](https://github.com/cabinetshah2710/Operating-system-project-on-Banker-s-Algorithm/tree/master)