OPEN ENDED LAB

LAB # 8

Spring 2020

CSE204L Operating Systems Lab

Submitted by: **Shah Raza**

Registration No. : **18PWCSE1658**

Class Section: **B**

"On my honor, as student of University of Engineering and Technology, I have neither given nor received unauthorized assistance on this academic work."

Student Signature: _____

Submitted to:

Engr. Mian Ibad Ali Shah

Monday, July 27th, 2020

Department of Computer Systems Engineering

University of Engineering and Technology, Peshawar

Task :

Using previous knowledge of OS theory/ Lab, develop/ simulate any process scheduling algorithm (except that I have covered in the lab), preferably Round Robin Scheduling. You may also work on thread scheduling/ deadlock avoidance.

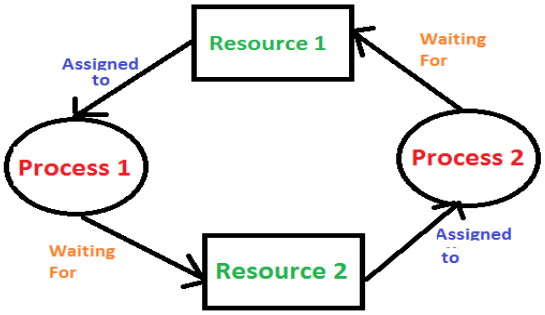
Problem Analysis:

I have decided to go for the Deadlock Avoidance as my Open Ended Lab task. In the following sections I will explain what is deadlock, how to avoid it and write code for it.

Deadlock:

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.

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 Avoidance:

There are several solutions for deadlock but we are going to discuss a method called deadlock avoidance. 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 in order to avoid deadlock.

Banker’s Algorithm:

Banker’s Algorithm is resource allocation and 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.

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.

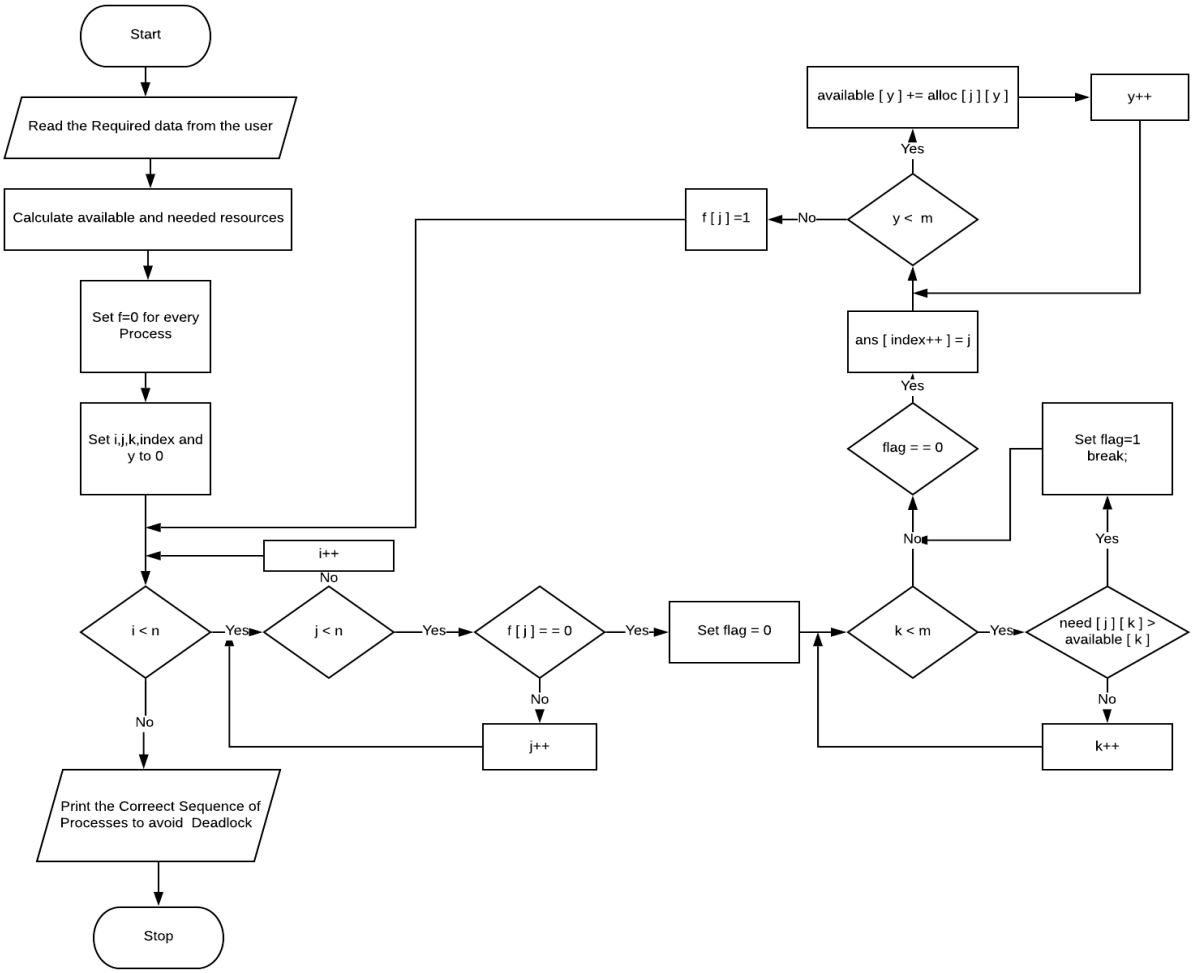
Allocation :

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]

Flowchart *:*



Code:

```
#include <stdio.h>
int main()
{
	int num_p,num_r,i,j,k;
	printf("Enter the number of Processes: ");
	scanf("%d",&num_p);
	printf("Enter the number of Resources: ");
	scanf("%d",&num_r);
	char ch='A';
	int resource_instance[num_r];
	for(i=0;i<num_r;i++)
	{
		printf("Enter the number of instances of Resource %c: ",ch);
		scanf("%d",&resource_instance[i]);
		ch+=1;
	}

	int alloc[num_p][num_r];
	for(i=0;i<num_p;i++)
	{
		ch='A';
		printf("Enter the Allocated Instances of Resources for P%d:\n",i);
		for(j=0;j<num_r;j++)
		{
			printf("No of Instances of Resource %c Allocated to P%d: ",ch,i);
			scanf("%d",&alloc[i][j]);
			ch+=1;
```

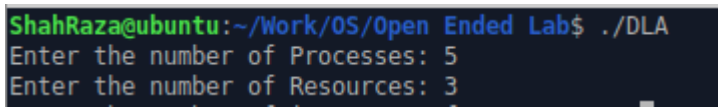
```
        }
}
int max[num_p][num_r];
for(i=0;i<num_p;i++)
{
        ch='A';
        printf("Enter the Maximum Instances of Resources for P%d:\n",i);
        for(j=0;j<num_r;j++)
        {
        printf("Maximum No of Instances of Resource %c Allowed to P%d: ",ch,i);
                scanf("%d",&max[i][j]);
                ch+=1;
        }
}
int available[num_r];
for(j=0;j<num_r;j++)
{
        int sum=0;
        for(k=0;k<num_p;k++)
                sum+=alloc[k][j];
        available[j]=resource_instance[j]-sum;
}
int f[num_p];
for(i=0;i<num_p;i++)
        f[i]=0;
int need[num_p][num_r];
for(i=0;i<num_p;i++)
{
```

```
        for(j=0;j<num_r;j++)
                need[i][j]=max[i][j]-alloc[i][j];
}
int index=0,y,ans[num_p];
for(i=0;i<num_p;i++)
{
        for(j=0;j<num_p;j++)
        {
                if(f[j]==0)
                {
                        int flag=0;
                        for(k=0;k<num_r;k++)
                        {
                                if(need[j][k]>available[k])
                                {
                                        flag=1;
                                        break;
                                }
                        }
                        if(flag==0)
                        {
                                ans[index++]=j;
                                for(y=0;y<num_r;y++)
                                        available[y]+=alloc[j][y];
                                f[j]=1;
                        }
                }
        }
```

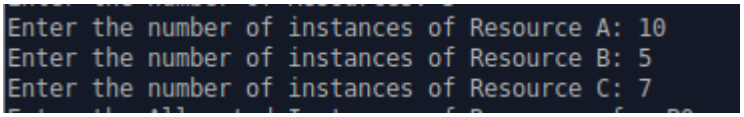
```
        }
        printf("Following is the safe sequence to Avoid Dead-Lock: \n");
        for(i=0;i<num_p-1;i++)
                printf("P%d -->",ans[i]);
        printf("P%d\n",ans[num_p-1]);
        return 0;
}
```

Inputs:

No of Processes and Resources:

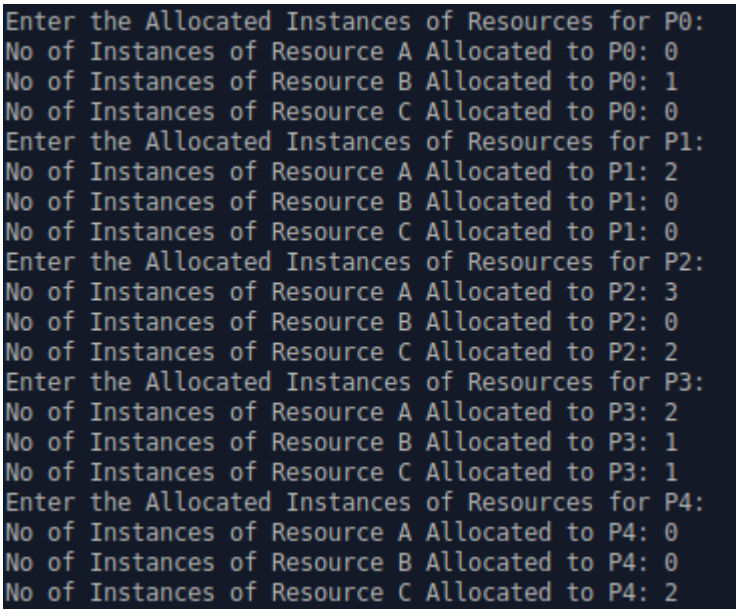


Instances of Resources:



Resource type A has 10 instances, B has 5 instances and type C has 7 instances.

Allocated Resources:



Maximum Instances of Resources by Different Processes:

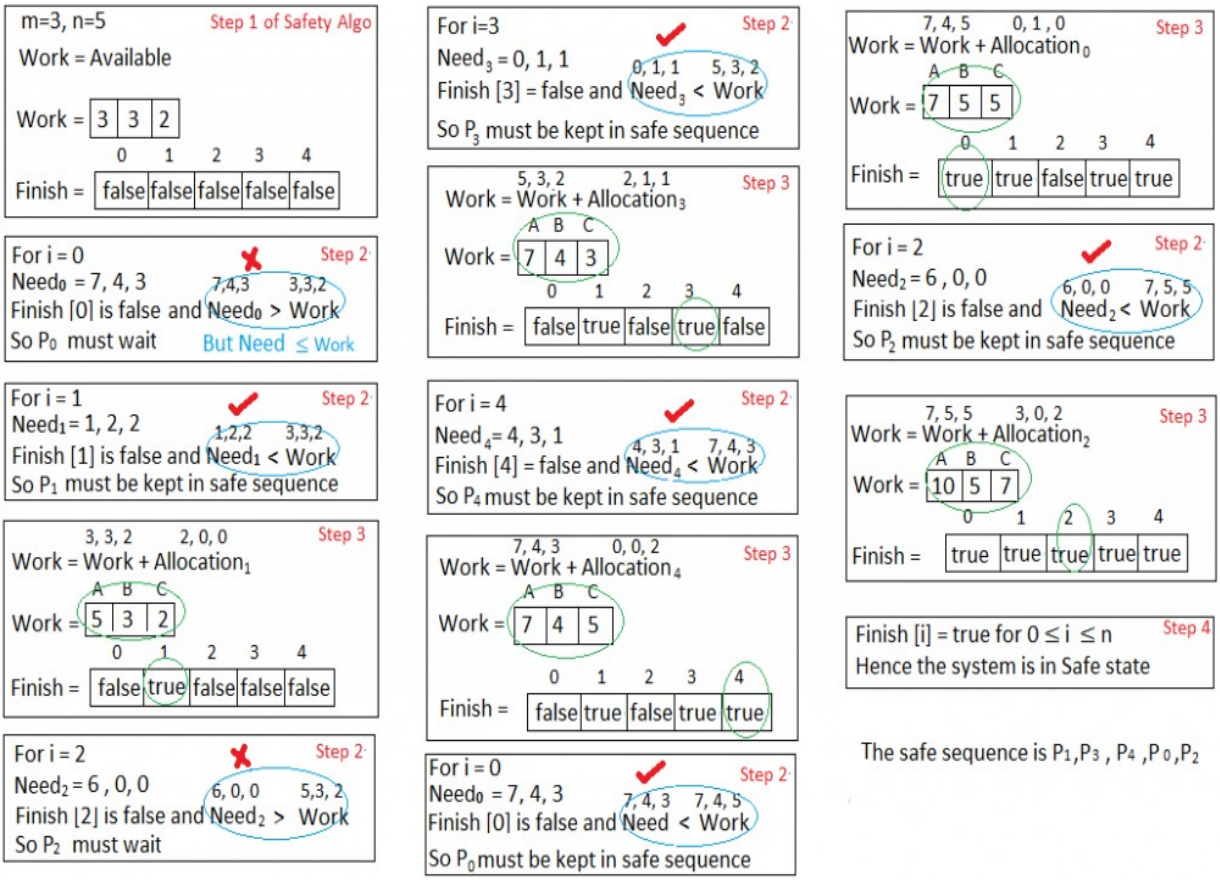
```
Enter the Maximum Instances of Resources for P0:
Maximum No of Instances of Resource A Allowed to P0: 7
Maximum No of Instances of Resource B Allowed to P0: 5
Maximum No of Instances of Resource C Allowed to P0: 3
Enter the Maximum Instances of Resources for P1:
Maximum No of Instances of Resource A Allowed to P1: 3
Maximum No of Instances of Resource B Allowed to P1: 2
Maximum No of Instances of Resource C Allowed to P1: 2
Enter the Maximum Instances of Resources for P2:
Maximum No of Instances of Resource A Allowed to P2: 9
Maximum No of Instances of Resource B Allowed to P2: 0
Maximum No of Instances of Resource C Allowed to P2: 2
Enter the Maximum Instances of Resources for P3:
Maximum No of Instances of Resource A Allowed to P3: 2
Maximum No of Instances of Resource B Allowed to P3: 2
Maximum No of Instances of Resource C Allowed to P3: 2
Enter the Maximum Instances of Resources for P4:
Maximum No of Instances of Resource A Allowed to P4: 4
Maximum No of Instances of Resource B Allowed to P4: 3
Maximum No of Instances of Resource C Allowed to P4: 3
```

Background Processing:

Process	Allocation	Max	Available
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Need [i, j] = Max [i, j] – Allocation [i, j]
So, the content of Need Matrix is:

Process	Need		
	A	B	C
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1



Output / Result:

```
Following is the safe sequence to Avoid Dead-Lock:
P1 -->P3 -->P4 -->P0 -->P2
ShahRaza@ubuntu:~/Work/OS/Open Ended Lab$
```

Discussion and Conclusion:

As we can see from the output, the only way to avoid deadlock in our example is to run P1 first, followed by P3, P4, P0 and at last P2. If we run the processes in any other order we will find ourselves in a deadlock situation.