OPERATING SYSTEMS

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

TOPIC: DEADLOCK DETECTION IN DISTRIBUTED SYSTEM

SUBMITTED FOR THE COURSE: OPERATING SYSTEMS (CSE2005) BY

TEAM MEMBERS:

1. VARSHA S 19BCE2346

2. DESAMSETTI CHARAN 19BCE0586

3. PAAVAN SATYA 19BCE0543

SLOT-F1

NAME OF THE FACULTY: PRIYA M

CONTENTS

Chapter Title

ABSTRACT:

In this project we have worked on deadlock detection in distributed systems. We have implemented three algorithms for deadlock detection. They are:

- Fully distributed deadlock distributed algorithm
- Centralized approach for deadlock detection
- Message parsing algorithm for deadlock detection

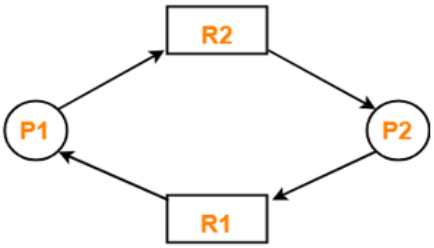
INTRODUCTION:

What Are DEADLOCKS?

A deadlock is a situation where two computer programs which are sharing the same resource are effectively preventing each other from accessing that resource which results in both the programs ceasing to function.

The earlier computer operating systems used to run only one program at a time. All the resources of the computer system were available to that one program. Later on the operating systems ran multiple programs at once, interleaving them. Programs were required to specify in advance what all resources they needed so that conflicts could be avoided with the other programs running at the same time. Eventually some of the operating systems also offered dynamic allocation of the resources. Programs could then request the further

allocations of those resources after they had begun running. This gave rise to the problem of deadlocks.



Example of a deadlock

Causes Of Deadlocks

The following 4 conditions are the necessary conditions for the occurrence of deadlock-

1. Mutual Exclusion
2. Hold and Wait
3. No preemption
4. Circular wait

1. Mutual Exclusion

- There should exist at least one resource in the system which can be used by only one process at a time.
- If there exists no such resource, then deadlock can never occur.

- Printer is an example of a resource that can be used by only one process at a time.

2. Hold and Wait

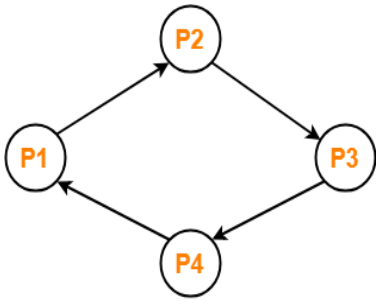
There must exist a process which holds some resource and waits for another resource held by some other process.

3. No Preemption

- Once the resource has been allocated to the process then it cannot be preempted.
- It means that the resource cannot be snatched forcefully from one process and given to the other process.
- The process should release the resource voluntarily by itself

4. Circular Wait

- All the processes must wait for the resource in a cyclic manner where last process waits for the resource which is held by the first process.



Circular Wait

LITERATURE SURVEY:

The algorithms used for deadlock detection are:

- **Fully distributed deadlock detection algorithm**

In the fully distributed deadlock detection approach, each site of the system shares equal responsibility for deadlock detection. In the algorithm, each site maintains its own local WFG. However to model waiting situations that involve external (nonlocal) processes, a slightly modified form of WFG is used. In this modified WFG an extra node is added to the local WFG of each site and also this node is connected to the WFG of the corresponding site. Now these modified WFGs are used for deadlock detection in the following manner. If a local WFG contains a cycle that does not involve the extra node, there is a possibility of a distributed deadlock that involves processes of multiple sites. To confirm a distributed deadlock, a distributed deadlock detection algorithm is invoked by the site whose WFG contains the cycle involving the extra node.

A problem associated with the above algorithm is that two sites may initiate the deadlock detection algorithm independently for a deadlock that involves the same processes. The result will be that both sites will update their local WFG s and search for cycles. After detecting a deadlock, both may initiate a recovery procedure that may result in killing more processes than is actually required to resolve the

deadlock. Furthermore, this problem also leads to extra overhead in unnecessary message transfers and duplication of deadlock detection jobs performed at the two sites.

- **Centralized approach for deadlock detection**

In the centralized deadlock detection approach, there is a local coordinator at each site that maintains a Wait For Graph (WFG) for its local resources, and there is a central coordinator (also known as centralized deadlock detector) that is responsible for constructing the union of all the individual WFGs. The central coordinator constructs the global WFG from information received from the local coordinators of all the sites. In this approach, deadlock detection is performed as follows:

i) If a cycle exists in the local WFG of any site, it represents a local deadlock. Such deadlocks are detected and resolved locally by the local coordinator of the site.
ii) Deadlocks that involve resources at two or more sites get reflected as cycles in the global WFG. Therefore, such deadlocks are detected and then resolved by the central coordinator.

In this centralized approach, the local coordinators send the local state information to the central coordinator in the form of messages.

Although the centralized deadlock detection approach is conceptually simple, it suffers from several drawbacks. First,

it is vulnerable to failures of the central coordinator. Second, the centralized coordinator can constitute a performance bottleneck in large systems having too many sites. Third, the centralized coordinator may detect false deadlocks.

- **Message parsing algorithm for deadlock detection**

Message Parsing algorithm for deadlock detection (Probe based distributed algorithm)

It is considered as the best algorithm. For detecting global deadlocks in distributed systems. This algorithm allows a process to request for multiple resources at one instance of time.

The algorithm is conceptually simple and works in the following manner. When a process that requests for a resources fails to get the requested resources and times out, it generates a special probe message and sends it to the process holding the requested resources. The probe message contains the following fields:

- The identifier of the process just blocked
- The identifier of the process sending this message.
- The identifier of the process to whom this message is being sent.

On receiving a probe message, the recipient checks to see if it itself is waiting for any resource. If not, this means that the recipient is using the resources requested by the process that sent the probe message to it. In this case, the recipient simply ignores the probe message. On the other hand, if the recipient

is waiting for any resource, it passes the probe message to the process holding the resource for which it is waiting. However, before the probe message is forwarded, the recipient modified its fields in the following manner:

1. The first field is left unchanged.
2. The recipient changes the second field to its own process identifier.
3. The third field is changed to the identifier of the process that will be the new recipient of this message.

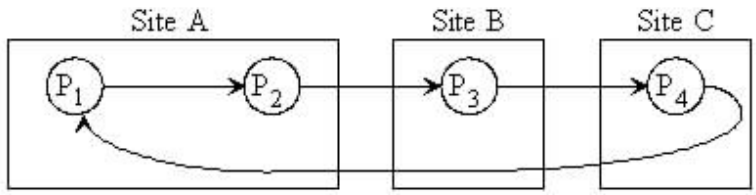
Very new recipient of the probe message repeats this procedure. If the probe message returns back to the original sender, a cycle exist and the system is deadlocked.

This algorithm is used in most distributed locking schemes due to the following attractive features of the algorithm:

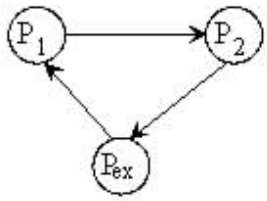
1. The algorithm is easy to implement, since each message is of fixed length and requires few computational steps.
2. The overhead of the algorithm is fairly low.
3. There is no graph constructing and information collecting involved.
4. False deadlocks are not detected by the algorithm.
5. It does not require any particular structure among the processes.

FULLY DISTRIBUTED DEADLOCK DETECTION ALGORITHM

Site A had some local processes that were involved in a deadlock cycle with some nonlocal processes

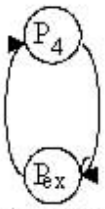


Actual global wait-for graph contains a cycle



Local wait-for graph at site A

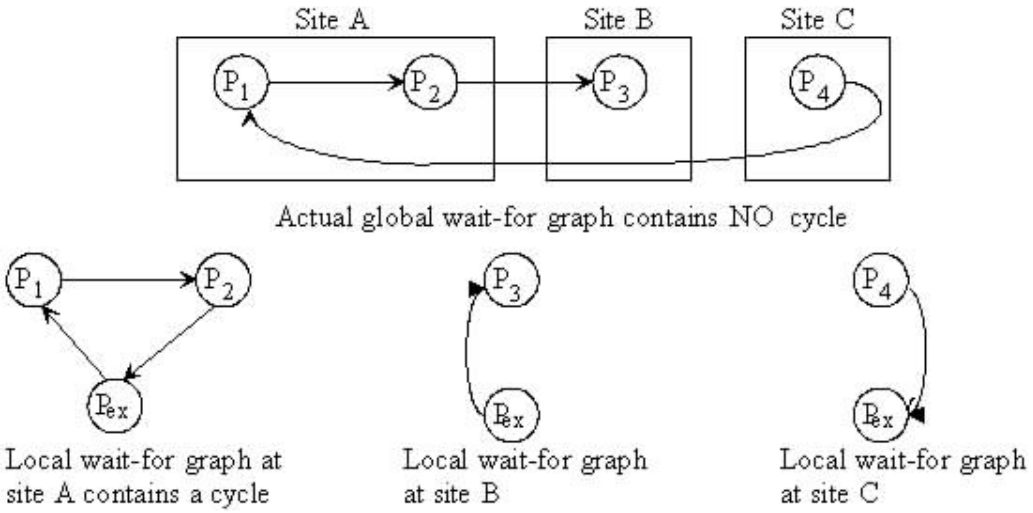
Local wait-for graph at site B



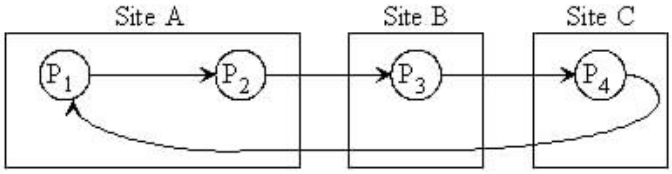
Local wait-for graph at site C

When the controller detects a cycle in the local wait-for graph containing only the local processes (excluding P_{ex}), then it has detected a deadlock.

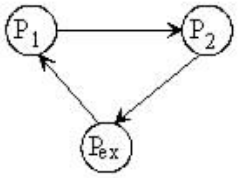
When a controller detects a cycle in the local wait-for graph containing P_{ex}, then it must invoke a distributed cycle-detection algorithm.



The controller detecting a cycle $\rightarrow P_{k_1} \rightarrow P_{k_2} \rightarrow \ldots \rightarrow P_{k_n} \rightarrow P_{ex}$ which contains P_{ex} of P_{ex} sends a message containing this path information to the site that contains the process for which P_{k_n} is waiting. That site then updates its wait-for graph with the information in the message and then it runs a cycle detection algorithm on the updated graph. If the updated wait-for graph detects a cycle containing P_e, then the procedure is repeated by sending a message containing the longer cycle to the next site. Deadlock cycles will then eventually be detected by some controller.

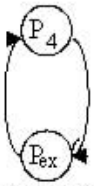


Actual global wait-for graph contains a cycle



Local wait-for graph
at site A

Local wait-for graph
at site B

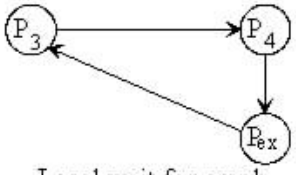


Local wait-for graph
at site C

Suppose B detects a cycle with P_e

Send cycle $P_e \rightarrow P_3 \rightarrow P_e$
to Site C

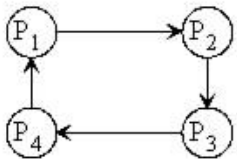
Send cycle $P_e \rightarrow P_3 \rightarrow P_4 \rightarrow P_e$
to Site A



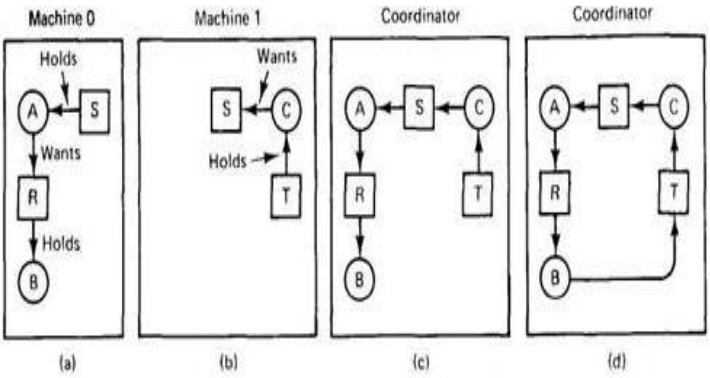
Local wait-for graph
at site C after updating

C detects a cycle with P_e

A detects deadlock



CENTRALIZED APPROACH FOR DEADLOCK DETECTION



(a) Initial resource graph for machine 0. (b) Initial resource graph for machine 1. (c) The coordinator's view of the world. (d) The situation after the delayed message.

Centralized:

One central site sets up a global WFG and then searches for cycles.

All the decisions are made by the central control node.

- It must maintain the global WFG constantly or
- Periodically reconstruct it.

The main advantage of this algorithm is that it permits the use of relatively simple algorithms.

The disadvantages of this algorithm include the following:

- There is one single point of failure.
- There can also be a communication bottleneck around the site due to all the WFG information messages.

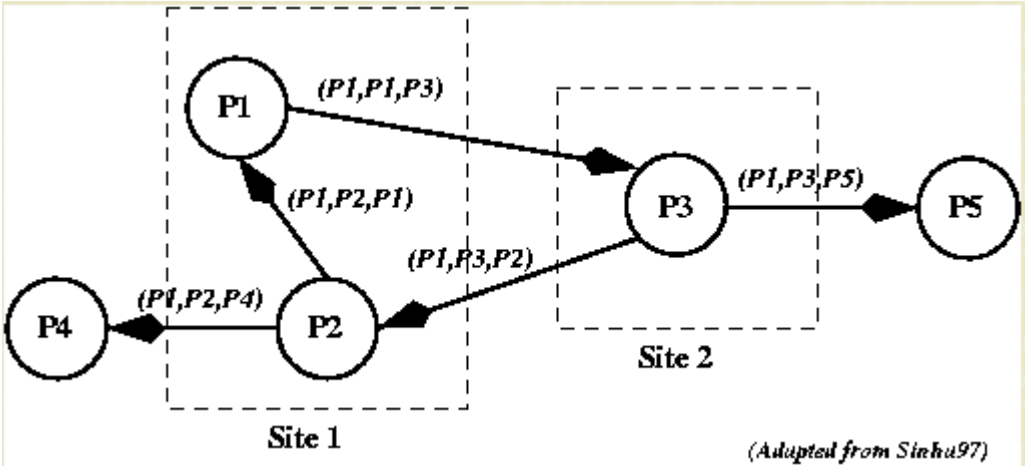
- Furthermore, this traffic is independent of the formation of any of the deadlocks.

Probe Based distributed Deadlock Detection Algorithm:

- This is considered an edge-chasing, probe-based algorithm. It is also considered as one of the best deadlock detection algorithms for distributed systems.
- If a process makes a request for a resource which fails or times out, the process generates a probe message and then sends it to each of the processes holding one or more of its requested resources.
- Each probe message contains the following information:
    1. the id of the process that is blocked (the one that initiates the probe message);
    2. the id of the process is sending this particular version of the probe message; and
    3. the id of the process that should receive this probe message.

- When a process receives a probe message, it also checks to see if it is also waiting for the resources.
- If not if it is currently using the needed resource and will then eventually finish and release the resource.
- If it is waiting for the resources, it passes on the probe message to all processes it knows to be holding resources it itself has requested.
- The process first modifies the probe message that it had received , changing the sender and the receiver ids.

- If a process receives a probe message that it recognizes as having initiated itself, it knows that there is a cycle in the system and thus, a deadlock exists.
- The following example is based on the same data used in the Silberschatz-Galvin algorithm example. In this case P1 initiates the probe message, so that all the messages shown have P1 as the initiator. When the probe message is received by process P3, it modifies it and sends it to two more processes. Eventually, the probe message returns to process P1. **Deadlock!**



CODE IMPLEMENTATION AND OUTPUT

```
#include<stdio.h>
void cda()
{
    int visited[10]={0,0,0,0,0,0,0,0,0,0};
int isCycle[10]={-1,-1,-1,-1,-1,-1,-1,-1,-1,-1};
typedef struct Graph
```

```
{
   int vertex;
   struct Graph *next;
}Graph;
Graph *g[10];
void initializeGraph(int n)
{
   int i;
   for(i=0;i<n;i++)
   {
      g[i]=(Graph*)malloc(sizeof(Graph));
      g[i]->vertex=i;
      g[i]->next=NULL;
   }
}
void addEdge(int v,int u)
{
   Graph *head=g[v];
   while(head->next)
      head=head->next;
   head->next=(Graph*)malloc(sizeof(Graph));
   head=head->next;
   head->next=NULL;
```

```
    head->vertex=u;
}
int checkCycle(int v,int n)
{
    Graph *head;
    int w;
    visited[v]=1;
    head=g[v]->next;
    while(head)
    {
        w=head->vertex;
        if(visited[w])
            return 1;
        if(isCycle[w] == -1)
                return checkCycle(w, n);          //We haven't visited that vertex yet
        else
            return 0;              //We visited this vertex before but a cycle was not found
        head=head->next;
    }
    visited[v]=0;
    isCycle[v]=0;
```

```
    return 0;
}
        int
n,nr[10],rag[10][10][10],np[10],wfg[10][10][10],rid[10][10],p
id[10][10],gwfg[10][10];
        int i,j,k,x,ver;
        int edg=0;
        printf("Enter the number of sites:");
        scanf("%d",&n);
        printf("Enter the total number of processes in the
distributed system:");
        scanf("%d",&ver);
        for(i=0;i<n;i++)
        {
            printf("\nEnter the number of resources available in
%d:",i+1);
            scanf("%d",&nr[i]);
            printf("\nEnter the number of process in available in
site %d:",i+1);
            scanf("%d",&np[i]);
        }
        for(i=0;i<n;i++)
        {
            for(j=0;j<nr[i];j++)
```

```
            {
                printf("\nIn site %d Enter the resource id of resource %d:",i+1,j+1);
                scanf("%d",&rid[i][j]);
            }
        }
        for(i=0;i<n;i++)
        {
            for(j=0;j<np[i];j++)
            {
                printf("\nIn site %d Enter the process id of process %d:",i+1,j+1);
                scanf("%d",&pid[i][j]);
            }
        }
        for(i=0;i<n;i++)
        {
        printf("\nEnter the resource allocation graph for site %d:",i+1);
        for(j=0;j<nr[i];j++)
        {
            for(k=0;k<np[i];k++)
            {
                scanf("%d",&rag[i][rid[i][j]][pid[i][k]]);
```

```
            }
            }
        }
for(i=0;i<n;i++)
{
        for(j=0;j<nr[i];j++)
        {
            for(k=0;k<np[i];k++)
            {
                for(x=0;x<np[i];x++)
                {
                if(rag[i][rid[i][j]][pid[i][k]]==1 &&
rag[i][rid[i][j]][pid[i][x]] == 2)
                {
                    printf("\nIn site %d process %d is
dependent on process %d:",i+1,pid[i][k],pid[i][x]);
                    wfg[i][pid[i][k]][pid[i][x]]=1;
                    gwfg[pid[i][k]][pid[i][x]]=1;
                    edg+=1;
                }
            }
        }
        }
```

```
}
for(i=0;i<10;i++)
{
	for(j=0;j<10;j++)
	{
		if(gwfg[i][j]==1)
		{
			printf("\nThere is a dependency from process %d to %d",i,j);
		}
		else
		{
			gwfg[i][j]=0;
		}
	}
}
   int a[100],v,u;
   initializeGraph(ver);
	for(i=0;i<5;i++)
	{
		for(j=0;j<5;j++)
		{
			if(gwfg[i][j]==1)
```

```
                        addEdge(i-1,j-1);
                    }
                }
    int z=checkCycle(0,ver);
    if(z==1)
    printf("\nCycle found\tDeadlock in the system");
    else
    printf("\nCycle not found\tSystem is deadlock free");
    printf("\n\n");
}

void fda()
{
        int
p[10],n,i,j,p1,s1,sp1,sp2,d0[10][10],d1[10][10],d2[10][10],l0[
10],l1[10],l2[10],x=0,y=0,z=0,c_matrix[15][15],a=0,b=0,c=0,
d=0,f[]={},s;
//to get No. of sites
printf("Enter total no. of sites\n");
scanf("%d",&n);
//No of processes in each site
for(i=0;i<n;i++)
{
printf("Enter total no. of process in S%d\n",i+1);
```

```
scanf("%d",&p[i]);
}
for(i=0;i<n;i++)
{
printf("Total no. of process in S%d are %d\n",i+1,p[i]);
}
printf("Enter the dependency matrix of processes of  Site1 connected with requesting process\n");
printf("Assume the ex node as 1, and x node is present in each site \n");
//dependancy matrix for site1
//dependancy matrix is computed by placing 1 in the matrix for processe when a process is dependant on other.
for(i=2;i<p[0]+2;i++)
{
   for(j=2;j<p[0]+2;j++)
   {
      scanf("%d",&d0[i][j]);
   }
}
//dependancy matrix for site2
for(i=5;i<p[1]+5;i++)
{
   for(j=5;j<p[1]+5;j++)
```

```
    {
        scanf("%d",&d1[i][j]);
    }
}
//dependancy matrix for site3
for(i=8;i<p[2]+8;i++)
{
    for(j=8;j<p[2]+8;j++)
    {
        scanf("%d",&d2[i][j]);
    }
}


//local cycle is computed for each site,when processes between the sites has dependancy between them
printf("The local cycle for the  sites are\n");
for(i=2;i<p[0]+2;i++)
{
    for(j=2;j<p[0]+2;j++)
    {
      if(d0[i][j]==1)
       {
```

```
            if(d0[j][j+1]==1)
            {
               while(x!=1){
               printf("The local cycle for the  site1 are\n");
               printf("1,%d,%d,%d,1\n",i,j,j+1);
               l0[a++]=i;
               l0[a++]=j;
               l0[a++]=j+1;
               x=1;
               }
        }
        }
    }
}
for(i=5;i<p[2]+5;i++)
{
    for(j=5;j<p[2]+5;j++)
    {
       if(d1[i][j]==1)
       {
          while(y!=1){
          if(d1[j][j+1]==1)
          {
```

```
            printf("The local cycle for the  site2 are\n");
          printf("1,%d,%d,%d,1\n",i,j,j+1);
            l1[b++]=i;
            l1[b++]=j;
            l1[b++]=j+1;
          y=1;
          }
          }
      }
    }
}
for(i=8;i<p[2]+8;i++)
{
   for(j=8;j<p[2]+8;j++)
   {
     if(d2[i][j]==1)
     {
        while(z!=1){
       if(d2[j][j+1]==1)
       {
          printf("The local cycle for the  site3 are\n");
        printf("1,%d,%d,%d,1\n",i,j,j+1);
         l2[c++]=i;
```

```
            l2[c++]=j;
            l2[c++]=j+1;
           z=1;
      }}
      }
   }
}
```

printf("Enter the connectivity matrix between 3 sites...Where the last node of each graph is connected to..\n");

//connectivity matrix between all sites. Connectivity matrix is computed to show how a site in one process is connected to other process.

```
for(i=4;i<11;i+=3)
{
   for(j=2;j<11;j++)
   {
      scanf("%d",&c_matrix[i][j]);
   }
}
```

//this part is for computing the path between site 1 and site2

```
printf("Path between Site1 and Site2 is\n");
for(i=4;i<8;i+=3)
{for(j=2;j<8;j++)
   {
```

```
        if(c_matrix[i][j]==1)

printf("1,%d,%d,%d,%d,%d,%d,1\n",l0[0],l0[1],i,j,l1[1],l1[2])
;
    }
}
//this part is for path between site2 and site3
printf("Path between  Site2 and Site3 is\n");
for(i=7;i<11;i+=3)
{
    for(j=8;j<11;j++)
    {
      if(c_matrix[i][j]==1)

printf("1,%d,%d,%d,%d,%d,%d,1\n",l1[0],l1[1],i,j,l2[1],l2[2])
;
    }
}
//path between all the sites
printf("Path between Site1, Site2 and Site3 is\n");
for(i=4;i<11;i+=3)
{
    for(j=2;j<11;j++)
    {
```

```
      if(c_matrix[i][j]==1)
      if(i==10){
printf("1,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,%d,1\n",l
0[0],l0[1],l0[2],l1[0],l1[1],l1[2],l2[0],l2[1],i,j,l0[1],l0[2]);
      f[d++]=j;
      f[d++]=j+1;
      f[d++]=j+2;
      f[d++]=j+3;
      f[d++]=j+4;
      f[d++]=j+5;
      f[d++]=j+6;
      f[d++]=j+7;
      f[d++]=i;
      f[d++]=j;
      f[d++]=j+1;
      f[d++]=j+2;
    }
    }
}
//if the final path contains a cycle then it leads to deadlock
s=(int)(sizeof(f)/sizeof(f[0]));
//printf("%d",f[p[0]+p[1]+p[2]]);
for(i=0;i<3;i++){
```

```
if(f[p[0]+p[1]+p[2]]==l0[i])
{
  printf("Deadlock detected... The final String contains a cycle ");
}
}
}
void mp()
{
      int p[10],n,i,p1,s1,sp1,sp2;
//Number of sites
printf("Enter total no. of sites\n");
scanf("%d",&n);
//Number of processes in each site.
for(i=0;i<n;i++)
{
printf("Enter total no. of process in S%d\n",i+1);
scanf("%d",&p[i]);
}
for(i=0;i<n;i++)
{
printf("Total no. of process in S%d are %d\n",i+1,p[i]);
}
```

```
//deadlock is computed if the two edges in probe message is same.i.e probe(i,j,k) if i and k are same it leads to deadlock.

//deadflock initiation point

printf("Enter the site no. and process id for which deadlock detection shold be initiated\n");

scanf("%d %d",&s1,&p1);

while(1)

{

printf("\nEnter the the processes of two different sites connected with requesting edge\n");

scanf("%d %d",&sp1,&sp2);

printf("\nProbe message is (%d,%d,%d)",p1,sp1,sp2);

if(p1==sp2)

{

printf("\nDeadlock detected");

break;

}

}

}

int main()

{

    int n;

    printf("1.Central Algorithm\n2.Fully distributed\n3.Message parsing");
```

```
        printf("\n Enter the choice");
        scanf("%d",&n);
        switch(n)
        {
                case 1: cda();
                        break;
                case 2: fda();
                        break;
                case 3: mp();
                        break;
                default : printf("\n invalid choice");
        }
}
```

OUTPUT:

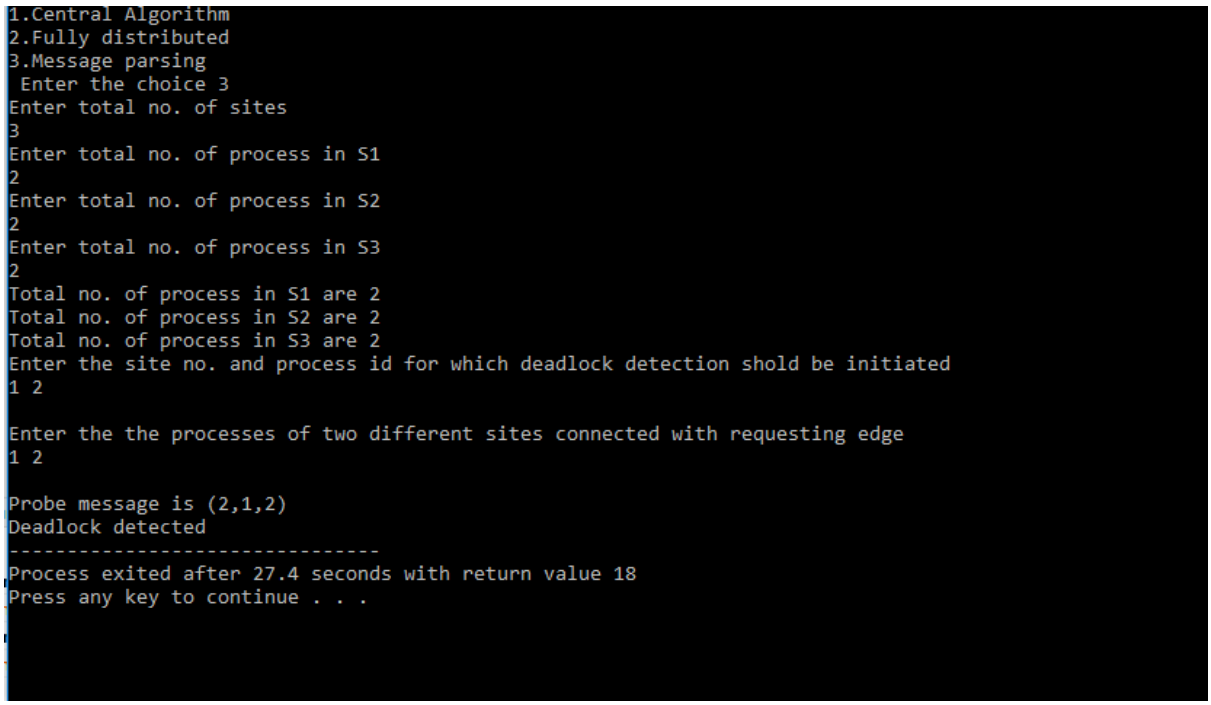
CENTRALIZED APPROACH

```
In site 1 Enter the process id of process 2:1

In site 1 Enter the process id of process 3:2

In site 2 Enter the process id of process 1:2

In site 2 Enter the process id of process 2:1

Enter the resource allocation graph for site 1:2
1
1
0
0
2

Enter the resource allocation graph for site 2:1
2

In site 1 process 1 is dependent on process 3:
In site 1 process 2 is dependent on process 3:
In site 2 process 2 is dependent on process 1:
There is a dependency from process 1 to 3
There is a dependency from process 2 to 1
There is a dependency from process 2 to 3
Cycle not found System is deadlock free


--------------------------------
Process exited after 186.2 seconds with return value 0
Press any key to continue . . .
```

FULLY DISTRIBUTED

```
1.Central Algorithm
2.Fully distributed
3.Message parsing
 Enter the choice2
Enter total no. of sites
3
Enter total no. of process in S1
3
Enter total no. of process in S2
3
Enter total no. of process in S3
3
Total no. of process in S1 are 3
Total no. of process in S2 are 3
Total no. of process in S3 are 3
Enter the dependency matrix of processes of  Site1  connected with requesting process
Assume the ex node as 1, and x node is present in each site
0 1 0
0 0 1
1 0 0
0 1 0
0 0 1
1 0 0
0 1 0
0 0 1
1 0 0
The local cycle for the  sites are
The local cycle for the  site1 are
1,2,3,4,1
The local cycle for the  site2 are
1,5,6,7,1
The local cycle for the  site3 are
1,8,9,10,1
Enter the connectivity matrix between 3 sites...Where the last node of each graph is connected to..
0
0
0
1
0
0
0
0
0
0
0
0
0
0
0
1
```

MESSAGE PARSING



REFERENCE:

- Distributed Operating Systems
  -By Pradeep K Sinha