VIT®

Vellore Institute of Technology

(Deemed to be University under section 3 of UGC Act, 1956)

DISTRIBUTED OPERATING SYSTEM

Project Report
on

Implementing Deadlock Detection and Avoidance System among various application running on several computing system

Submitted By-

Hrishikesh S G (22MCA0162)
Rajat Singh (22MCA0139)
Sambit Basu (22MCA0240)

Guided By- Dr. SENTHILKUMAR T

DECLARATION

We hereby declare that the report entitled "**Implementing Deadlock Detection and Avoidance System among applications running on several computing system**" submitted by me, for the ITA5006 Distributed Operating Systems (EPJ) to Vellore Institute of Technology is a record of bonafide work carried out by me under the supervision of **Dr. SENTHILKUMAR T.**

We further declare that the work reported in this report has not been submitted and will not be submitted, either in part or in full, for any other courses in this institute or any other institute or university.

Place : Vellore

Date :

Signature of Candidates

Table of Contents

Table of Figures

ABSTRACT

Sharing and assigning resources among running and new processes is an operating system task. In a multiprocessing climate, assignment of resources can turn into a point of conflict. Such a struggle might lead the system into a state known as deadlock. The extent of the topic is to recognize the circumstances for deadlock, recognize the various conditions that lead to this undesirable state, and distinguish the strategies forrecover of the deadlock system, prevention, and the detection of the deadlock.

The distributed system set is primarily involved by huge association for their features. Whenever we foster a deadlock detection and its recovery approaches for a distributed system. In a distributed system a deadlock is a condition where an interaction can't continue since it requires an available resource held by another cycle yet it itself is holding a resource that the other system needs. Similar circumstances for stops in uniprocessors apply to distributed system. As the distributed system is so vast that the deadlock can neither be prevented nor avoided so the occurrence of deadlock should be controlled effectively by their detection and resolution. Sometimes it leads to serious system failures. After implementation of detection algorithm, the deadlock is resolve by deadlock resolution algorithm and there are basically three approaches to detect deadlocks in distributed system they are Centralized approach, Distributed approach, and Hierarchical approach.

1. INTRODUCTION

In an operating system, there are many processes (processes are programs that are currently being executed) running continuously. All of these processes are important for the functioning of the computer. It defines the basic unit of work that has to be implemented by the system. Processes follow sequential execution. These processes require some resources to run or finish their execution. Sequential order is followed here. The process first requires a resource. If it is available, the OS grants the resource. If not, the process must wait. After the process has used the resource, it is released. If a problem arises during the waiting and releasing, it means that there is a deadlock situation.

A deadlock is a common situation in operating systems where a process waiting for a resource can be executed because that resource is currently held by another process and is being utilized for its execution, therefore, the process does not get executed. Moreover, many other processes may also be waiting for a resource to be released. A deadlock situation is created.

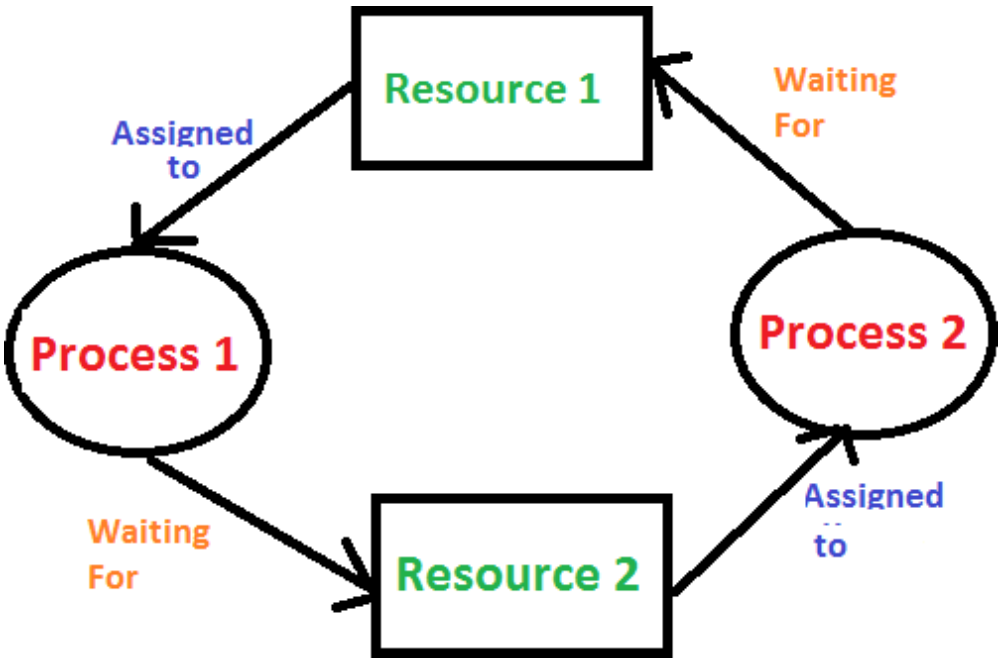

Fig 1.1 Wait for graph

1.1 Deadlock Conditions

Situations of Deadlock Processes do not operate continuously from the time they are formed until they are terminated; they are classified as three states ready, running, or blocked. A process is halted in the ready state to allow another process to run; in the running.

state, a process is utilizing some resource; and in the blocked state, a process is stopped and will not restart until it will get a desired output. Deadlock is a common undesired circumstance in which two processes are in a running state, have obtained some resources, but need to trade resources to continue. Both processes arewaiting for the other to deliver a "requested"

resource that neither of them will ever provide; as a result, neither of them is progressing, resulting in a stalemate. The Dinning Philosopher's Problem is one of many scenarios that have been created to explain stalemate situations.

1.2Necessary Conditions for Deadlock

MUTUAL EXCLUSION CONDITION:

The resources can't be shared.

Explanation: At least one resource (thread) must be in a non-shareable mode, which means that only one process can claim exclusive ownership of the resource at any given time. If another process asks the resource, the requesting process must wait until the resource is released before proceeding.

1. **HOLD AND WAIT CONDITION**:

The request process is currently halted, and resources are awaiting requests. Explanation: A process must exist that is holding a resource that has already been allocated to it while it waits for additional resources that are currently held by other processes.

2. NO-PREEMPTIVE CONDITION:

Preempting resources that have already been allocated to a process is not possible. Explanation: Resources cannot be removed from processes until they are completed or voluntarily released by the process that holds them.

3. CIRCULAR WAIT CONDITION:

The system's processes form a circular list or chain, with each process in the chain waiting for a resource held by the process after it.

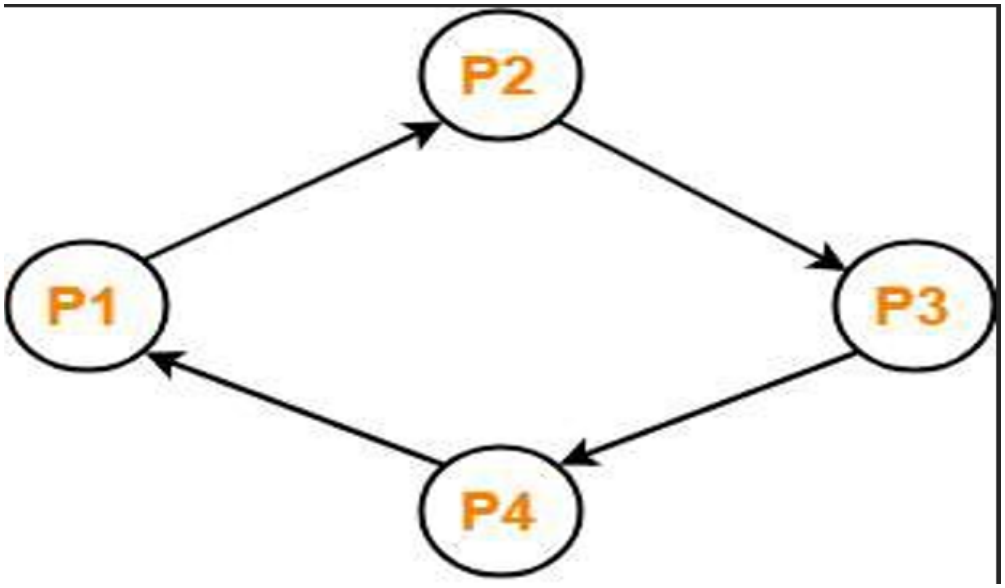

Fig 1.2 Circular Wait

1.3 Deadlock Prevention

Deadlocks are an undesirable state of the system, and it is necessary to prevent or detect them, as well as recover from them if they are unavoidable. There are a variety of methods and approaches available, and some operating systems completely ignore the problem; in these cases, deadlock detection and recovery become the primary focus. Predicting potential deadlock conditions is difficult, but there are some general condition that, if met, ensure that deadlocks will occur.

If all of the following conditions are met, for example:

a. mutual exclusion condition

b. no preemption condition.

c. a hold-and-wait condition.

d. a circular-wait condition.

Some solutions to provide some type of deadlock prevention mechanism for more "pro-active" operating systems have been proposed.

1.4 Deadlock Recovery

Deadlock recovery can be accomplished in a variety of methods. One typical method is to set priorities, which includes pushing a process to give up a resource so other asking processes can move on. This solution is contingent on the system's complexity and the ability to pick and re-allocate resources. Unfortunately, when the process that forcibly released a holding resource re-acquires it and resumes its task as if nothing had happened, the problem of continuation reappears. However, by adding checkpoint system scheduling, 'snapshots' of the state of the system at various instances become available, enabling for deadlock recovery based on resource release as indicated by just a pre-deadlock state ‘snapshot.' It is also possible to return to previous instances and resume the lower priority ones. Recovery usually starts from a previous checkpoint, which means some progress is lost. Surprisingly, if a lower priority process seeks the resource that was previously given up, it will return to the wait state. Deadlock recovery is most efficient when one of the deadlocked processes is terminated. It's a harsh and unforgiving method of breaking deadlocks that necessitates making a rational decision about which process should be terminated.

The decision to terminate the process should be based on the nature of the process and the potential losses associated with its termination. Unfortunately, because there can be and over two deadlock processes, there is no rule for what constitutes a reasonable termination. Processes that can be repeated and yield the same outcome are called repeatable processes.

2. Literature Survey

Table 2.1 Literature Survey Table

PAPER NUMBER	PAPER SURVEY
1	In this paper the author depicted a decentralized methodology for impact evasion, gridlock location, and stop goals among different versatile robots which pursue freely arranged directions. The author presented a blend of three completely disseminated calculations which dependably settle the undertaking. They don't utilize any worldwide synchronization, don't meddle with one another, and request just neighborhood entomb robot correspondence. The worldwide coordination of a fleet of robots is accomplished by utilizing just nearby coordination of sets of robots. The plan to accomplish this is to permit more than one coordination connect for every robot. These connections a lot of robots together in a worldwide control structure. Halts in the worldwide coordination, which can't be stayed away from when just neighborhood coordination is utilized, are dependably distinguished. The stops are settled by altering the course of coordination connections and requesting that robots plan elective directions. The main halts which can't be settled are those where the direction arranging layers of the included robots are not ready to design elective directions. This, in any case, depends entirely on the capacities of the direction arranging layer and the properties of the earth. The strict division of direction arranging and crash evasion/gridlock taking care of makes it conceivable to utilize many direction organizers with our framework. The main confinement is, that they ought to have the capacity to design elective directions. There are no other unique suppositions about the earth, or the application made. As an end, it can say that our methodology is entirely appropriate to facilitate various autonomously moving robots through neighborhood entomb robot correspondence in a ton of uses. Our reproduction results confirm this. For further work, it as intriguing to explore distinctive techniques for the development of the execution ways of the TCDs, the execution way would be built such that circumstances favored, clearly more gridlocks could be settled by changing edge bearings and less halts would need to be settled by replanning directions. Moreover, extraordinary procedures for the crash shirking could be assessed.
2	In this paper the author tends to discuss the solutions of a deadlock. So a deadlock occurs when there is a set of processes waiting for resources held by other processes in the same set. The processes in deadlock wait indefinitely for the resources and never terminate their executions and the resources they hold are not available to any other process. The techniques to deal with deadlock such as deadlock avoidance, prevention etc. but still deadlock can occur. The only way to deal with deadlock when it occurs is to detect and resolve it as soon as possible. Several techniques to resolve deadlock are mentioned above. One can use any of the above technique to resolve deadlock and deadlock will be resolved.

3	In this paper the author research on the different pattern of distributed deadlock scheduling and the impact of deadlock detection scheduling on the system performance and show that there exists an optimal deadlock detection frequency that yields the minimum long run mean average cost associated with the message complexity of deadlock detection. The key element approach is to develop a time-dependent model that associates the deadlock resolution cost with the deadlock persistence time. It assists the study of time-dependent deadlock resolution cost in connection with the rate of deadlock formation and the frequency of deadlock detection initiation, differing significantly from the past research that focuses on minimizing per-detection and per-resolution costs.
4	In this paper the focus is of mobile agents which are process which can move from one host to another host. The processes called by client are executed on a machine. The host machine must have all the resource needed to implement the service. There are many areas expanding the middleware due to the implementation of application-level protocols for communication. In the case of distributed systems, the tasks to control the operating systems are tiresome; this is because database systems deadlock, mutual exclusion and concurrency control are difficult in case of centralized systems. As the processes have used resources at various sites to improve throughput distributed systems is more vulnerable to deadlock occurrence. Hence there is a need to resolve deadlock by proper resource allocation.
5	In this paper, the distributed deadlock detection algorithms are classified based on the underlying deadlock models such as AND model, OR model, AND-OR model and P out of Q model. Among the models, P out of Q model, also called as generalized request model, has the modeling power of all other models and much more concise expressive power than other models. However, it is difficult to detect deadlock in the generalized request model. It is observed that only very few algorithms have been proposed to detect deadlock in the literature. Though the algorithms have reduced the deadlock duration through the years, they have paid very little attention on other key measures such as number of messages and message size.

3. Architecture

We are involving “Wait for Graph” and “Path pushing Algorithm” to address the Deadlock and to identifying distributed environment. To distinguish stop and evasion and goal. We are involving Wait for graph procedure and Path push to identify deadlock and Banker's algorithm for the avoidance.

1) WFG_-> It is one of the finest algorithms for detecting the deadlock system in distributed operating system and it is suitable where the databases are small in size. In this method, a graph is drawn **primarily based totally at the** transaction and their lock **at the** resource. If the graph created has a closed-loop or a cycle, then **there's** a deadlock.

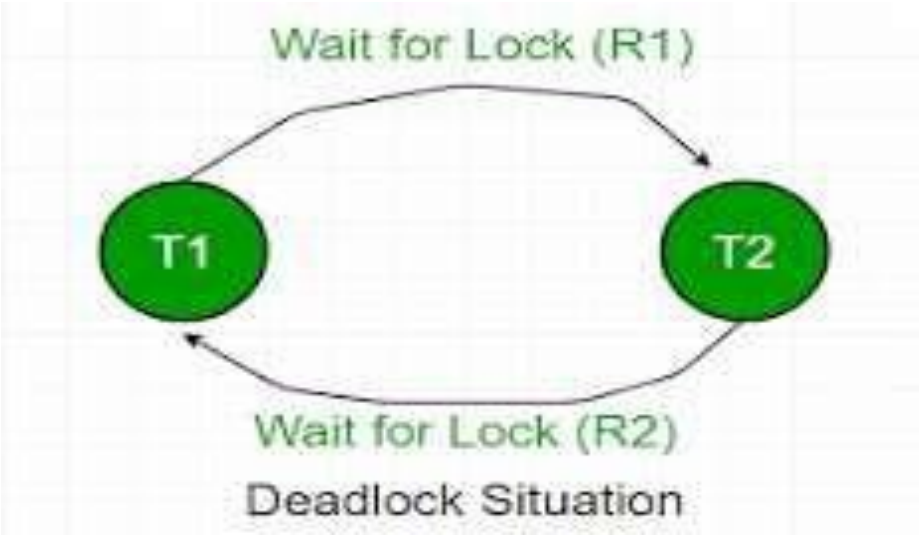

Fig 3.1 Deadlock Situation

Algorithm:

Step 1: Take the primary process (Pi) from the useful resource allocation graph and check the route wherein it is acquiring aid (Ri) and start a wait-for-graph with that specific technique.

Step 2: Make a direction for the Wait-for-Graph in which there is probably no resource allocated from the contemporary process (Pi) to next process (Pj), from that subsequent process (Pj) discover an aid (Rj) that allows you to be received through next way (Pk) that is launched from machine (Pj).

Step 3: Repeat Step 2 for all the available processes in the distributed system.

Step 4: After final touch of all strategies, if we find out a closed-loop cycle then the machine is in a impasse state i.e.- the system is in deadlock state.

Below is the given example.

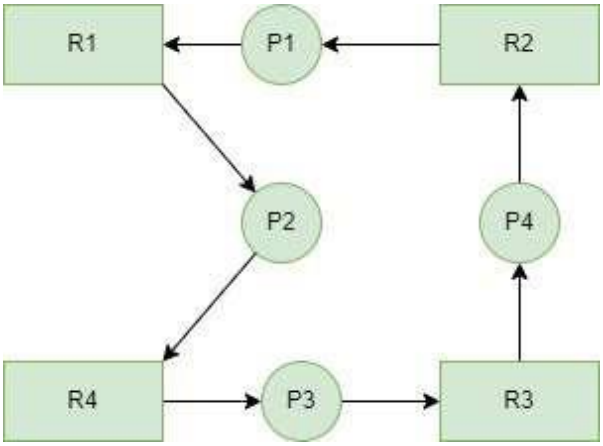

Fig 3.2 Deadlock Example

1. **Path Pushing Technique:**

This algorithm finds out allotted deadlocks by way of keeping a unique worldwide WFG. The principal concept is to create a global WFG for every distributed device connected in the system. During the computation of the deadlock, every site sends its WFG saved locally to its neighboring sites.

2. **Chandy-Misra-Haas's detection algorithm:**

- Another name of Chandy-Misra Haas's algorithm is Edge Chasing Algorithm
- In an edge-chasing algorithm, the presence of a cycle in a distributed graph structure is verified by propagating special messages called probes, along the edges of the graph.
- These probe messages are different than the request and reply to messages.
- The formation of cycle can be deleted by a site if it receives the matchingprobe sent by it previously.
- Whenever a process that is executing receives a probe message, it discards this message and continues.
- Only blocked processes propagate probe messages along their outgoing edges.

- Main advantage of edge-chasing algorithms is that probes are fixed size messages which is normally very short.

Algorithm:

Process of sending probe:

1. If process Pi is locally dependent on itself, then declare a deadlock.

2. Else for all Pj and Pk check following condition:

   (a) Process Pi is locally dependent on process Pj

   (b) Process Pj is waiting on process Pk

   (c) Process Pj and process Pk are on different sites.

If all the above conditions are true, send probe (i, j, k) to the home site of processPk:

On the receipt of probe (i, j, k) at home site of process Pk:

Process Pk checks the following conditions:

(a) Process Pk is blocked.

(b) Dependent k[i] is *false*.

(c) Process Pk has not replied to all requests of process Pj

If all the above conditions are found to be true then:

1. Set dependent k[i] to true.

2. Now, if k == i then, declare the Pi is deadlocked.

3. Else for all Pm and Pn check following conditions:

   (a) Process Pk is locally dependent on process Pm and

   (b) Process Pm is waiting upon process Pn and

   (c) Process Pm and process Pn are on different sites.

4. Send probe (i, m, n) to the home site of process Pn if above conditions satisfy.

Thus, the *probe* message travels along the edges of transaction wait-for (TWF) graph and when the *probe* message returns to its initiating process then it is said that deadlock has been detected.

Performance:

- Algorithm requires at most exchange of $m(n-1)/2$ messages to detect deadlock. Here, m is number of processes and n is the number of sites.
- The delay in detecting the deadlock is $O(n)$.

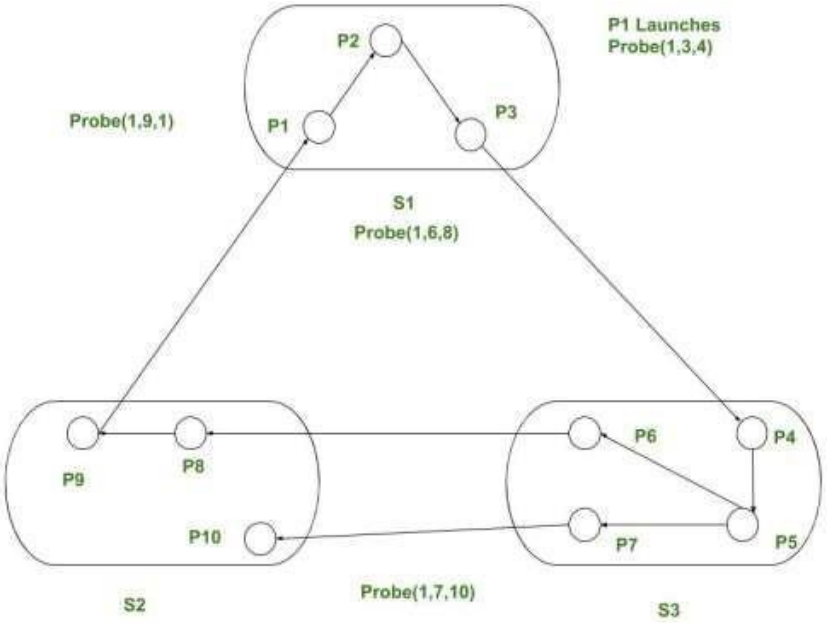

Fig 3.3 Chandy-Misra-Haas's distributed deadlock detection algorithm

4.Implementation

4.1 Code to understand Deadlock in Java:

```
public class DeadlockDemo {
    public static void main(String[] args){
        final String resource1 = "Techvidvan ";
        final String resource2 = "Java Tutorial";

        Thread t1 = new Thread() {
            public void run() {
                synchronized(resource1) {
                    System.out.println("Thread 1: Locked resource1");
                    try {
                        Thread.sleep(100);
                    }
                    catch(Exception e) {}
                    synchronized(resource2) {
                        System.out.println("Thread 1: Locked resource2");
                    }
                }
            }
        };
        Thread t2 = new Thread(){
            public void run()
            {
                synchronized(resource1){
                    System.out.println("Thread 2: Locked resource1");
                    try{
                        Thread.sleep(100);
                    }
                    catch(Exception e){}
                    synchronized(resource2) {
                        System.out.println("Thread 2: Locked resource2");
                    }
                }
            }
        };

        t1.start();
        t2.start();
    }
}
```

Result/Output

4.2 Bankers Algorithm Code

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.

```
import java.util.Scanner;

// create BankersAlgoExample class to implement Banker's algorithm in Java
class BankersAlgo
{

    // create findNeedValue() method to calculate the need of each process
    static void findNeedValue(int[][] needArray, int[][] maxArray, int[][]
allocationArray, int totalProcess, int totalResources)
    {
        // use nested for loop to calculate Need for each process
        for (int i = 0 ; i < totalProcess ; i++){    // for each process
            for (int j = 0 ; j < totalResources ; j++){  //for each resource
                needArray[i][j] = maxArray[i][j] - allocationArray[i][j];
            }
        }
    }

    // create checkSafeSystem() method to determine whether the system is in safe state or
not
    static boolean checkSafeSystem(int[] processes, int[] availableArray, int[][]
maxArray, int[][] allocationArray, int totalProcess, int totalResources)
    {
```

```
        int [][]needArray = new int[totalProcess][totalResources];

        // call findNeedValue() method to calculate needArray
        findNeedValue(needArray, maxArray, allocationArray, totalProcess, totalResources);

        // all the process should be infinished in starting
        boolean []finishProcesses = new boolean[totalProcess];

        // initialize safeSequenceArray that store safe sequenced
        int []safeSequenceArray = new int[totalProcess];

        // initialize workArray as a copy of the available resources
        int []workArray = new int[totalResources];

        //use for loop to copy each available resource in the workArray
        System.arraycopy(availableArray, 0, workArray, 0, totalResources);

        // initialize counter variable whose value will be 0 when the system is not in the
safe state or when all the processes are not finished.
        int counter = 0;

        // use loop to iterate the statements until all the processes are not finished
        while (counter < totalProcess)
        {
            // find infinished process which needs can be satisfied with the current work
resource.
            boolean foundSafeSystem = false;
            for (int m = 0; m < totalProcess; m++)
            {
                if (!finishProcesses[m])        // when process is not finished
                {
                    int j;

                    //use for loop to check whether the need of each process for all the
resources is less than the work
                    for (j = 0; j < totalResources; j++)
                        if (needArray[m][j] > workArray[j])      //check need of current
resource for current process with work
                            break;

                    // the value of J and totalResources will be equal when all the needs
of current process are satisfied
                    if (j == totalResources)
                    {
                        for (int k = 0 ; k < totalResources ; k++)
                            workArray[k] += allocationArray[m][k];

                        // add current process in the safeSequenceArray
                        safeSequenceArray[counter++] = m;
```

```
                        // make this process finished
                        finishProcesses[m] = true;

                        foundSafeSystem = true;
                    }
                }
            }

            // the system will not be in the safe state when the value of the
foundSafeSystem is false
            if (!foundSafeSystem)
            {
                System.out.print("The system is not in the safe state because lack of
resources");
                return false;
            }
        }

        // print the safe sequence
        System.out.print("The system is in safe sequence and the sequence is as follows:
");
        for (int i = 0; i < totalProcess ; i++)
            System.out.print("P"+safeSequenceArray[i] + " ");

        return true;
    }

    // main() method start
    public static void main(String[] args)
    {
        int numberOfProcesses, numberOfResources;

        //create scanner class object to get input from user
        Scanner sc = new Scanner(System.in);

        // get total number of resources from the user
        System.out.println("Enter total number of processes");
        numberOfProcesses = sc.nextInt();

        // get total number of resources from the user
        System.out.println("Enter total number of resources");
        numberOfResources = sc.nextInt();

        int[] processes = new int[numberOfProcesses];
        for(int i = 0; i < numberOfProcesses; i++){
            processes[i] = i;
        }

        int[] availableArray = new int[numberOfResources];
        for( int i = 0; i < numberOfResources; i++){
```

```
            System.out.println("Enter the availability of resource"+ i +": ");
            availableArray[i] = sc.nextInt();
        }

        int[][] maxArray = new int[numberOfProcesses][numberOfResources];
        for( int i = 0; i < numberOfProcesses; i++){
            for( int j = 0; j < numberOfResources; j++){
                System.out.println("Enter the maximum resource"+ j +" that can be
allocated to process"+ i +": ");
                maxArray[i][j] = sc.nextInt();
            }
        }

        int[][] allocationArray = new int[numberOfProcesses][numberOfResources];
        for( int i = 0; i < numberOfProcesses; i++){
            for( int j = 0; j < numberOfResources; j++){
                System.out.println("How many instances of resource"+ j +" are allocated to
process"+ i +"? ");
                allocationArray[i][j] = sc.nextInt();
            }
        }

        //call checkSafeSystem() method to check whether the system is in safe state or
not
        checkSafeSystem(processes, availableArray, maxArray, allocationArray,
numberOfProcesses, numberOfResources);
    }
}
```

Result/Output

```
Enter total number of processes
5
Enter total number of resources
3
Enter the availability of resource0:
3
Enter the availability of resource1:
2
Enter the availability of resource2:
7
Enter the maximum resource0 that can be allocated to process0:
5
Enter the maximum resource1 that can be allocated to process0:
3
Enter the maximum resource2 that can be allocated to process0:
3
Enter the maximum resource0 that can be allocated to process1:
2
Enter the maximum resource1 that can be allocated to process1:
2
Enter the maximum resource2 that can be allocated to process1:
9
Enter the maximum resource0 that can be allocated to process2:
0
Enter the maximum resource1 that can be allocated to process2:
2
Enter the maximum resource2 that can be allocated to process2:
2
```

```
Enter the maximum resource0 that can be allocated to process3:
2
Enter the maximum resource1 that can be allocated to process3:
4
Enter the maximum resource2 that can be allocated to process3:
3
Enter the maximum resource0 that can be allocated to process4:
3
Enter the maximum resource1 that can be allocated to process4:
0
Enter the maximum resource2 that can be allocated to process4:
1
How many instances of resource0 are allocated to process0?
0
How many instances of resource1 are allocated to process0?
2
How many instances of resource2 are allocated to process0?
0
How many instances of resource0 are allocated to process1?
0
How many instances of resource1 are allocated to process1?
3
How many instances of resource2 are allocated to process1?
0
How many instances of resource0 are allocated to process2?
2
How many instances of resource1 are allocated to process2?
2
```

```
How many instances of resource1 are allocated to process2?
2
How many instances of resource2 are allocated to process2?
1
How many instances of resource0 are allocated to process3?
1
How many instances of resource1 are allocated to process3?
0
How many instances of resource2 are allocated to process3?
0
How many instances of resource0 are allocated to process4?
2
How many instances of resource1 are allocated to process4?
2
How many instances of resource2 are allocated to process4?
2
The system is in safe sequence and the sequence is as follows: P2 P3 P4 P0 P1
Process finished with exit code 0
```

4.3 Code For Chandy Misra

```
      import java.io.*;

public class ChandyMisraHaas
{
    public static int flag=0;
    public static void main(String[] args)throws Exception
    {
        BufferedReader ob=new BufferedReader(new InputStreamReader(System.in));
        int init,aa,bb,x=0,end=5;

        File input=new File("Dependencies.txt");
        BufferedReader in=new BufferedReader(new InputStreamReader(new
FileInputStream(input)));
        String line;int[][] a=new int[end][end];
        line=in.readLine();line=in.readLine();
        while((line=in.readLine())!=null)
        {aa=3;bb=4;
            for(int y=0 ; y<end ; y++)
            {
                a[x][y]=Integer.parseInt(line.substring(aa,bb));
                aa+=2;bb+=2;
            }
            x++;
        }


System.out.println("_____________________________________________________________");Syst
em.out.println();
        System.out.println(" CHANDY-MISRA-HAAS DISTRIBUTED DEADLOCK DETECTION
ALGORITHM");System.out.println();
        System.out.println("\tS1\tS2\tS3\tS4\tS5");
        for(int i=0 ; i<end ; i++)
        {
            System.out.print("S"+(i+1)+"\t");
            for(int j=0 ; j<end ; j++)
            {
                System.out.print(a[i][j]+"\t");
            }
            System.out.println();
        }

        System.out.println();System.out.print("Enter Initiator Site No. : ");
        init=Integer.parseInt(ob.readLine());
        int j=init-1;

        System.out.println();System.out.println();
        System.out.println(" DIRECTION\tPROBE");System.out.println();

        for (int k=0 ; k<end; k++)
        {
            if(a[j][k]==1)
            {
                System.out.println(" S"+(j+1)+" --> S"+(k+1)+"
("+init+","+(j+1)+","+(k+1)+")");
                aman(a,j,k);
            }
        }
        if(flag==0){System.out.println();System.out.println(" NO DEADLOCK DETECTED");}

System.out.println("_____________________________________________________________");
```

```
        ob.readLine();

    }

    public static void aman(int[][] a,int init,int k)
    {int end=5;
        for(int x=0 ; x<end ; x++)
        {
            if(a[k][x]==1)
            {
                if(init==x)
                {
                    System.out.println(" S"+(k+1)+" --> S"+(x+1)+"
("+(init+1)+","+(k+1)+","+(x+1)+")"+" --------> DEADLOCK DETECTED");
                    flag=1;break;
                }
                System.out.println(" S"+(k+1)+" --> S"+(x+1)+"
("+(init+1)+","+(k+1)+","+(x+1)+")");
                aman(a,init,x);
            }
        }
    }

}
```

Result /Output

```
 CHANDY-MISRA-HAAS DISTRIBUTED DEADLOCK DETECTION ALGORITHM

    S1  S2  S3  S4  S5
S1  0   1   0   0   0
S2  0   0   1   0   0
S3  0   0   0   1   1
S4  0   0   0   0   0
S5  0   0   0   0   0

Enter Initiator Site No. : 1


 DIRECTION  PROBE

 S1 --> S2     (1,1,2)
 S2 --> S3     (1,2,3)
 S3 --> S4     (1,3,4)
 S3 --> S5     (1,3,5)

 NO DEADLOCK DETECTED

------------------------------------------------------------
```

5. Conclusion:

Deadlocks are an undesirable but intriguing system condition. There are multiple variable conditions that can lead to deadlocks and as a result make them very complex and difficult to prevent. Recovering from deadlocks also has its own challenges, particularly because it is difficult to reinstate the system, to a similar one prior to deadlock, in order to complete interrupted processing. There are some formalized methods via resource trajectories and state diagrams, which can to some extent predict deadlocks. However, completely restricting their occurrence requires a firmly designed operating system along with very efficient synchronization algorithms.

6. REFERENCES

[1] Pooja Chahar, Surjeet Dalal, "*Deadlock Resolution Technique: An Overview*", International Journal of Scientific and Research Publications, Volume 3, Issue 7, July 2013.

[2] Sahiba Raza, Jameel Ahmad, "*Study of Diverse Models of Deadlock Detection in Distributed Environment*", International Journal of Technical Reasearch and applications, Vol 6,Issue 1, PP. 56-59 Jan-Feb 2018.

[3] Alireza Soleimany, Zahra Giahi, "A*n Efficient Distributed Deadlock Detection and Prevention Algorithm by Daemons*", IJCSNS International Journal of Computer Science and Network Security, Vol.12 No4, April 2012.

[4] Sakshi Surve, Sarat Bhatt, "*A Modified Algorithm for Distributed Deadlock Detection in Generalized Model*", IOSR Journal of Engineering, Vol 6, Issue 4, PP 1-5, April 2016.

[5] Dr. Deepti Malhotra, "*Different Deadlock Handling Strategies in Distributed Environment*", International Journal of advanced Research in Computer Science and Software Engineering, Vol 6, Issue 2, February 2016.

[6] CHENG XIN, YANG XIAOZONG, "*A Concurrent Distributed Deadlock Detection/Resolution Algorithm for Distributed Systems*", Proceedings of the 5th WSEAS/IASME Int. Conf. on SYSTEMS THEORY and SCIENTIFIC COMPUTATION, Malta, September 15-17, 2005 (pp336-341)

[7] Aakriti Bhardwaj, Aastha Kapoor, "*Approaches For Deadlock Detection And Deadlock Prevention For Distributed Systems*", IJIRT, Vol 1, Issue 6, 2014.