



VIT[®]
Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

SCHOOL OF INFORMATION
TECHNOLOGY AND ENGINEERING

SUBJECT: OPERATING SYSTEM
PROJECT J-COMPONENT

REGISTER NUMBER: 21MCA0216 -RAHUL GUJARATHI

REGISTER NUMBER: 21MCA0213 -SUKESH THAPA

REGISTER NUMBER: 21MCA0027 -VINEETHA

REGISTER NUMBER: 21MCA0096 -TALHA

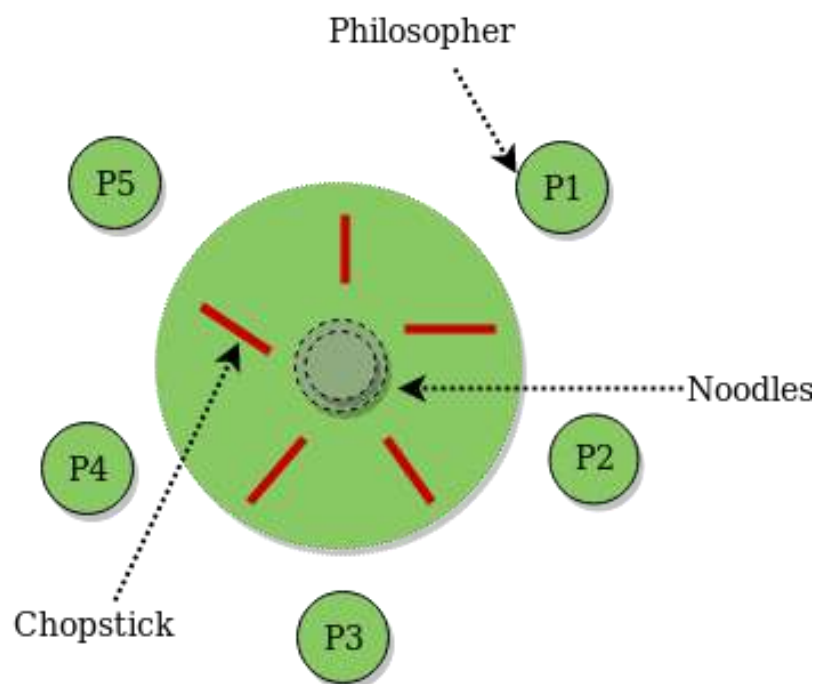
REGISTER NUMBER: 21MCA0201 -SRINIVAS

TITLE:

DINNING PHILOSOPHER'S PROBLEM WITH SOLUTION AND DEAD LOCK SOLUTION.

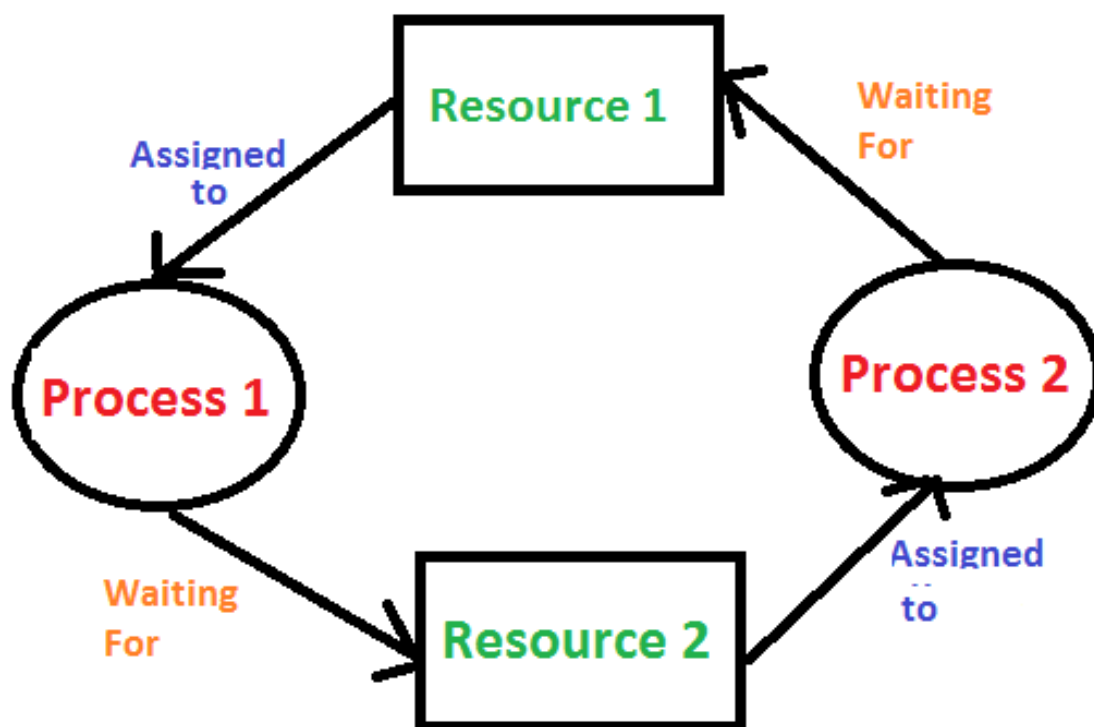
ABSTRACT:

The Dining Philosophers' problem is a classic synchronization problem which is used to evaluate situations where there is a need of allocating multiple resources to multiple processes. The five philosophers sit at a circular table with a large bowl of spaghetti in the center. A fork is placed in between each pair of adjacent philosophers. Each philosopher can only use the forks on his immediate left and immediate right. The philosophers never speak to each other, which creates a dangerous possibility of deadlock when every philosopher holds a left fork and waits perpetually for a right fork (or vice versa). We need to allocate the resources as in chopsticks to each philosopher and hence, avoid deadlock or starvation.



Dead Lock Resolution Techniques:

A deadlock occurs when there is a set of processes waiting for resource 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 occurrence of deadlocks should be controlled effectively by their detection and resolution, but may sometimes lead to a serious system failure. After implying a detection algorithm the deadlock is resolved by a deadlock resolution algorithm whose primary step is to either select the victim then to abort the victim. This step resolves deadlock easily. This paper describes deadlock detection using wait for graph and some deadlock resolution algorithms which resolves the deadlock by selecting victims using different criteria.



PROJECT OVERVIEW:

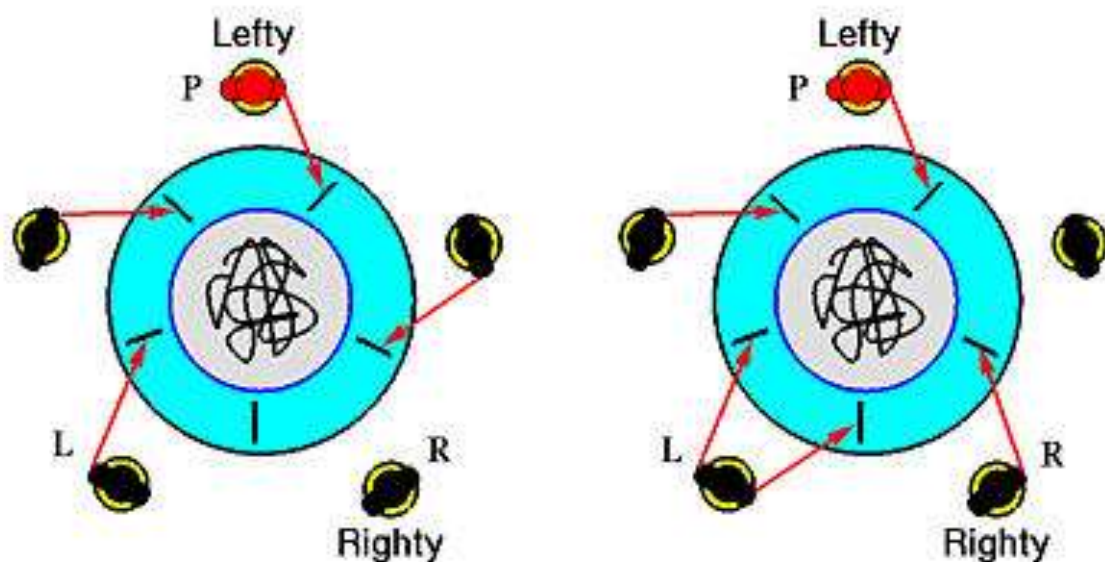
Five philosophers are sitting in a round table with a bowl of rice in the center. Each philosopher either thinks or eats. There is a single fork in between each of the philosophers. If a philosopher wants to eat, they grab the forks adjacent to them. Before a philosopher decides to eat, they must ensure both forks are available for use before eating. If both forks are available, the philosophers can eat and puts down the forks when finished and starts thinking again. Otherwise, they must wait until the forks are available.

EXISTING APPROACH:

The resource hierarchy solution involves assigning a partial order to the resources (forks). The resources must be claimed in a certain order. Also, no two resources unrelated by order can be used by a single process (philosopher) simultaneously. If there are 5 forks labelled 1 to 5, each philosopher will always grab the lower-numbered fork first, followed by the highernumbered fork. The order in which they put down the fork does not matter. As an example, if 4 of 5 philosophers grabbed their respective low-number fork, then only the highestnumber fork will remain. The 5th philosopher will not be able to grab a fork. This solution does avoid deadlock. However it is not practical, as the required resources are not known in advance and can become highly inefficient. The arbitrator solution introduces a waiter in the scene. If a philosopher is hungry, then it must ask the waiter for permission to eat. Permission is only given to one philosopher at a time until that philosopher picks up both forks. The con of this solution is reduced parallelism. If a philosopher is eating and one of the neighbors ask permission to eat, the other philosophers have to wait until that neighbor's request is granted by the waiter even if there are forks still available for the other philosophers.

OUR-APPROACH FOR THIS PROBLEM:

We will be implementing a solution where we will use both P-threads mutex locks and conditional variables. We will make sure that a philosopher will only eat if both forks are available aka picking them up in the critical region. There will be 5 philosophers created and each philosopher will be assigned a thread. A function `pickup_forks()` will be invoked whenever a philosopher wished to eat and philosophers will be alternating between thinking and eating. When the philosopher is finished eating, the forks will be returned using the `return_forks()` method. Conditional variables are used as a locking mechanism for data integrity with the help of P-threads mutex locks. The mutex locks will be responsible to allow access to the shared data for the threads.



DEAD LOCK RESOLUTIONS:

A deadlock occurs when there is a set of processes waiting for resource 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. A deadlock lowers the system utilization and hinders the progress of processes. Also the presence of deadlocks affects the throughput of the system. The dependency relationship among processes with

respect to resources in a distributed system is often represented by a directed graph, known as the Wait for Graph (WFG). In the WFG each node represents a process and an arc is originated from a process waiting for a resource to a process holding the resource. In a distributed system, a deadlock occurs when there is a set of processes and each process in the set waits indefinitely for the resources from each other. Therefore it is quite essential that a fast deadlock detection and resolution mechanism is applied otherwise the processes involved in the deadlock will wait indefinitely and will lower the system utilization and hinders the progress of processes. A deadlock needs to be resolved timely because if not resolved, the deadlock size will increase with the deadlock persistence time as more processes will be trapped in the deadlock where a deadlock size is defined as the total number of blocked processes BP) involved in deadlock, where BP is the process that waits indefinitely on other processes. Because of deadlock none of the any processes involved can make any progress without obtaining the resources for which they are waiting.

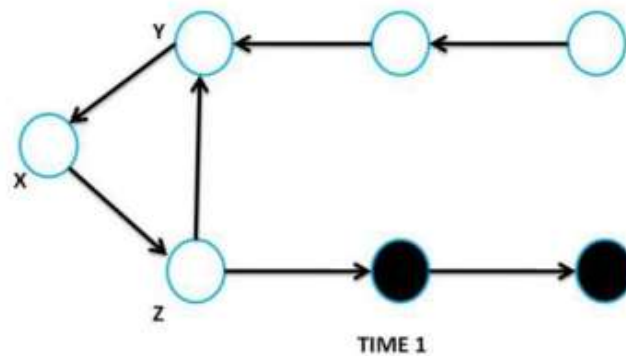


Figure1: A few processes in deadlock

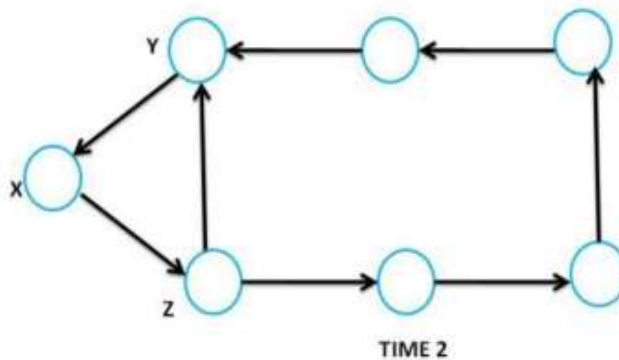


Figure 2: Increasing deadlock size as more processes trapped in deadlock

Figure1: A few processes in deadlock Figure 2: Increasing deadlock size as more processes trapped in deadlock Because distributed systems are vulnerable to deadlocks, the problems of deadlock detection and resolution have long been considered important problem in such systems. Several models have been proposed for the processes operating in distributed system. As per the AND model, a process sits idle until all of the requested resources are acquired. In the OR model, a process resumes execution if any of the requested resources is granted. In the P-out-of-Q model also known as the generalized model, a process

makes Q resource requests and remains blocked until it obtains any P resources. A generalized model is found in many domains such as resource allocation in distributed operating systems and communicating processes. A deadlock is defined differently depending on the underlying model. Since a process becomes blocked if any of its resource requests is not granted, a deadlock in the AND model corresponds to a cycle in the WFG. In the OR model, the presence of a knot in the graph implies a deadlock. In the generalized model a deadlock involves a more complex topology in the WFG. A cycle is a necessary but not sufficient condition for deadlock in this model.

WAIT FOR GRAPH: Deadlock detection is the process of actually determining that a deadlock exists and identifying the processes and resources involved in the deadlock. The basic idea is to check allocation against resource availability for all possible allocation sequences to determine if the system is in deadlocked state. Of course, the deadlock detection algorithm is only half of this strategy. Once a deadlock is detected, there needs to be a way to recover several alternatives exists:

- Temporarily prevent resources from deadlocked processes.
- Back off a process to some check point allowing preemption of a needed resource and restarting the process at the checkpoint later.
- Successively kill processes until the system is deadlock free. These methods are expensive in the sense that each iteration calls the detection algorithm until the system proves to be deadlock free. The complexity of algorithm is $O(N^2)$ where N is the number of proceeds. Another potential problem is starvation; same process killed repeatedly. The simplest and easiest way to detect deadlock is wait for graph. A wait-for graph in computer science is a directed graph used for deadlock detection in operating systems and relational database systems. In computer science, a system that allows concurrent operation of multiple processes and locking of resources and which does not provide mechanisms to avoid or prevent deadlock must support a mechanism to detect deadlocks and an

algorithm for recovering from them. One such deadlock detection algorithm makes use of a wait-for graph to track which other processes a process is currently blocking on. A wait for graph is a graph that consists of set of edges (E) and vertices (V). Processes are represented by vertices. In a wait-for graph, an edge from process P_i to P_j implies P_j is holding a resource that P_i needs and thus P_i is waiting for P_j to release its lock on that resource. A deadlock exists if the graph contains any cycles. The wait for graph scheme is applicable to a resource allocation system with multiple instances of each resource type.

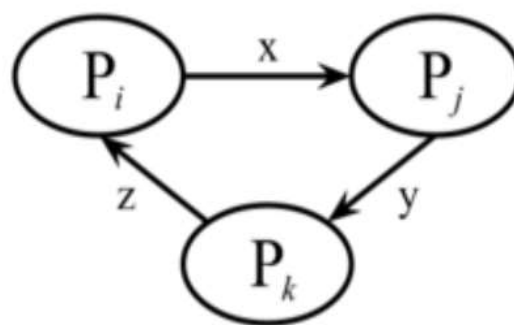


Figure 3: Wait for graph for deadlock detection.

Deadlock detection is the process of actually determining that a deadlock exists and identifying the processes and resources involved in the deadlock. The basic idea is to check allocation against resource availability for all possible allocation sequences to determine if the system is in deadlocked state. The deadlock detection and resolution algorithm always require that transactions should be aborted .For this reason several issues must be carefully considered.

- 1) Aborts are more expensive than waits.
- 2) Unnecessary aborts result in wasted system resources.
- 3) Optimal concurrency requires that the number of aborted transactions be minimized These factors must be considered so that the transaction being aborted will have the least impact on system performance and throughput. Basically the deadlocks present in a system are detected by a periodic initiation of an effective

deadlock detection algorithm and then resolved by a deadlock resolution algorithm and it is always tried that the resolution algorithm used does not cause any unnecessary aborts / roll backs. The appropriate scheme for handling deadlocks in distributed systems is detection and resolution. A typical method to resolve deadlock is to select a proper victim. The victim is to abort itself for deadlock resolution. The primary issue of deadlock resolution is to selectively abort a subset of processes involved in the deadlock so as to minimize the overall abortion cost. This is often referred to as the minimal abort set problem. The victim (aborted) processes need to cancel all pending requests and releases all acquired resources so that false deadlocks detection and resolution could be avoided. Usually, the deadlocks are resolved by aborting deadlocked processes. Therefore, two facts have to be considered when analyzing the cost associated to deadlock resolution algorithms: the cost of detecting a deadlock and the time that the aborted processes have wasted. Deadlock situations when detected should be resolved as soon as possible but ensuring a minimum number of abortions and only those processes should be aborted which has been selected as victim. Thus, algorithms (saferesolution algorithms) verifying the safety correctness criterion of resolving only true deadlocks should be designed In fact the deadlock detection using wait for graph is safe detection algorithm and it is considered correct because they detect in finite time, all deadlock of the system and do not detect false deadlock. Generally this algorithm doesn't take into account how a detected deadlock is resolved. It is only assumed that it is properly resolved. The algorithms do not explicitly model the resolution of detected deadlocks. Neither the system nor the code of the algorithm includes the effect of resolutions. Most of deadlock resolution algorithms abort or terminate the victim process.

RESOLUTIONALGORITHMS:

Here we will discuss two algorithms for deadlock resolution which uses different criterion for victim selection

A. Resolution by using Timestamp:

One of the most commonly used technique for deadlock resolution is timestamp based approach for selecting the victim. In this approach, a timestamp is allocated to each process as soon as it enters the system. The timestamp of the younger process is greater than the timestamp of older process. According to this approach, the victim is selected on this timestamps, the process with the higher timestamp is aborted, that is the youngest process is selected as the victim and is aborted in order to break the deadlock cycle. The goal behind choosing the youngest process as victim is that the youngest process would have used less resources and less CPU time as compared to older process. One problem with this technique is that it can cause starvation problem because every time a younger process is aborted which can starve the younger process from completion.

B. Resolution by using Burst time:

Another approach for selecting a victim to break deadlock cycle is considering the burst time of each process. Burst time means the CPU time needed by any process for its execution. This can also be considered as one parameter for selecting a victim. The process with maximum burst time can be aborted in order to break cycle. The problem with this technique is that it can abort the process with high burst time which has been in the system for very long i.e. an older process with high burst time can be aborted which is inefficient approach.

C. Resolution by Degree In a wait-for-graph:

for any system, the degree of any vertex denoting a process determines how many resources a process is holding and how many resources a process is requesting. There are two types of degrees in a directed WFG:

1. In-degree: In-degree means the number of edges coming to any node of WFG and it denotes number of request for resources held by a process.

2. Out-degree: Out-degree means the number of edges going out of a node in WFG denoting number of request for resources done by the node.

In resolution by degree, degree of each process is calculated and process having highest degree is aborted. Degree of any process can be calculated by taking sum of in-degree and out-degree.

D. Resolution by combination of Timestamp and Burst time:

Another approach for selecting victim for deadlock is using both timestamp and burst time in combination. Select a process as victim which is younger and has high burst time for resolving deadlock. The advantage with this approach is younger process which will take maximum execution time will be aborted to allow processes with less execution time to complete first.

E. Resolution by combination of Burst time and Degree:

Another combination for resolving deadlock is considering Burst time and Degree both for selecting a victim. Process with high burst time and high degree should be aborted that means a process which is having more resource request and will take high time to complete will be aborted. Although, there is still the problem of older process to be aborted but the advantage with this approach is aborting process with high burst time and high degree will release maximum resources needed for completion of other process with less execution time needed.

F. Resolution by combination of Degree and Timestamp:

Taking degree and timestamp both in combination for resolving deadlock can prove to be another technique for deadlock resolution. A younger transaction with high degree will be aborted. The problem of starvation in considering only timestamp will be avoided in this case as degree of the node is also considered

along with timestamp in order to select victim for resolving deadlock in the system.

G. Time Efficient Deadlock Resolution Algorithm Deadlock:

is a major concern in a distributed system, since resources are shared among processes at sites distributed across a network. One of the most accepted methods of deadlock handling is detection and resolution. Both deadlock prevention and avoidance strategies are conservative solutions, whereas deadlock detection is optimistic. In deadlock detection and resolution, deadlocks are allowed to occur. Periodically, or on certain conditions, a detection algorithm is executed; if any deadlock state is found, resolution is undertaken. To resolve a detected deadlock, the system must abort one or more processes involved in the deadlock and release the resources allocated to the aborted processes. Here deadlock resolution with reusable resources is considered. In resolving a deadlock state, it is desirable to minimize the number of processes to abort to make the system deadlock-free. Concept of release set is introduced here. A release set is a set of one or more processes that can be reduced if a process is aborted and its resources are released. The release set is represented by $R(p_i)$. For example, release set of process P_7 and P_5 in figure 4 is $R(P_7) = \{P_8\}$ and $R(P_5) = \{P_6, P_7, P_8\}$.

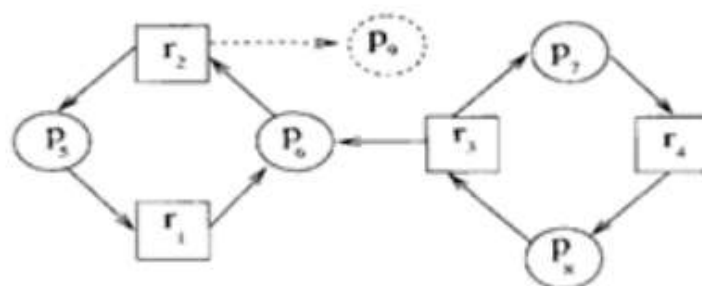


Figure 4: Deadlock cycle

Here two criteria for victim selection are considered. According to criterion 1 abort P such that $|R(p)| = \max\{|R(p_i)|, 1 \leq i \leq N_p\}$ where N_p is the number of processes that are reduced. Using Criterion 1 in the example of figure 4, P_6 or P_6 is chosen. The second criterion for deadlock resolution that is present concerns counting the number of cycles that involve resources held by a process i.e resources held by a process is involved in how many deadlock cycles. A deadlock vector d is defined such that $d_i = c$ represents that vertex i is involved in c different cycles. Now according to criterion 2 for each process in the RAG, we sum the numbers of cycles to which the resources held by the process are bound. A good victim candidate is then a process which maximizes deadlock vector. In figure 4, p_6 holds two resources, each of which are involved in one cycle. All other processes in figure 4 hold one resource, each of which are involved in one cycle. Thus, P_6 with $d_6 = 2$ would unambiguously be chosen by Criterion 2

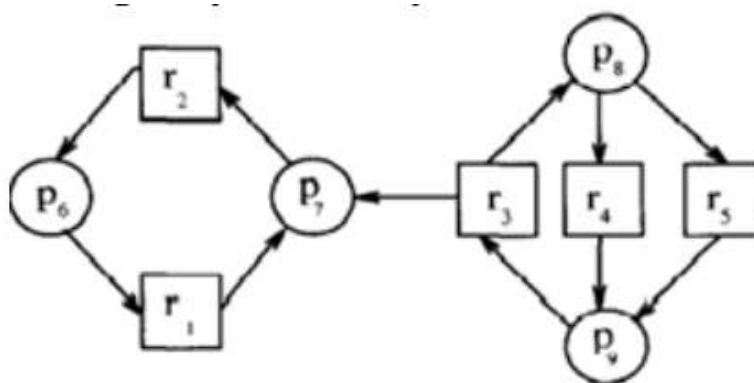


Figure 5: Deadlock Cycle

In figure 5, P_8 holds one resource which is involved in two cycles, P_9 holds two resources, each of which are involved in one cycle, and P_7 holds two resources, one which is involved in one cycle and one which is involved in two cycles. In other words, $d_7 = 2$ and $P_7 = 3$; thus, P_7 would be selected by Criterion 2. In both our example RAGS, only a single process is aborted by Criterion 2.

H. VGS Algorithm for Deadlock Resolution:

This section describes the solution to deadlocks in distributed systems i.e. VGS Algorithm an efficient deadlock resolution algorithm. In a distributed system if deadlock is detected at a site, then the site coordinator can apply VGS algorithm to resolve the deadlock[15]. This algorithm is based on the mutual cooperation of the transactions and is described as follows:

Suppose $T_i, T_{i+1}, T_{i+2}, \dots, T_n$ are the transactions involved in a deadlock. They form a deadlock cycle such that T_i holds resource R_i , T_{i+1} holds resource R_{i+1} , T_{i+2} holds resource R_{i+2}, \dots, T_n holds R_n and T_i is requesting for resource R_{i+1} , T_{i+1} is requesting for resource R_{i+2}, \dots, T_n is requesting for R_i . Since each transaction is holding a resource and waiting indefinitely for other resource held by the other.

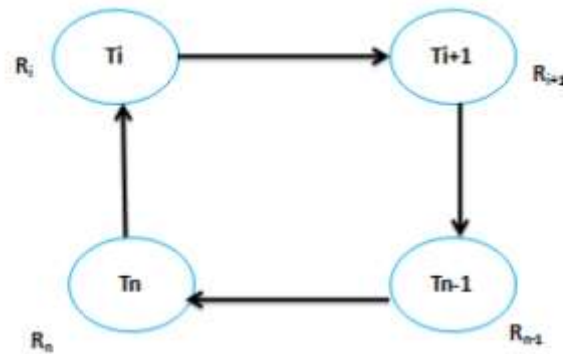
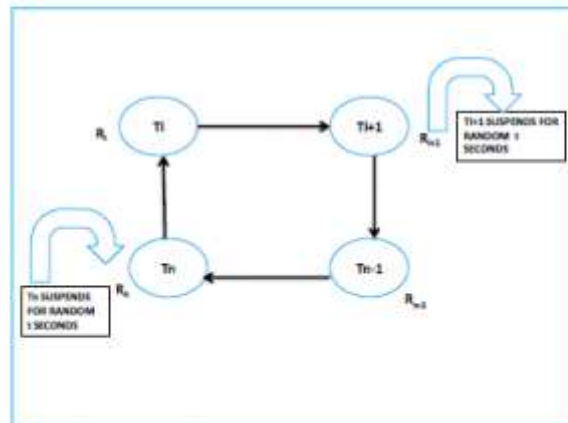


Figure 6: A deadlock cycle

T_i REQUESTS R_{i+1}
 T_{i+1} REQUESTS R_{i+2}
 \vdots
 T_{n-1} REQUESTS R_n
 T_n REQUESTS R_i



transaction, they form a deadlock cycle and none of them is being able to proceed ahead. In the proposed deadlock resolution algorithm transaction, coordinator observes the scenario and it suspends T_{i+1} for some random t seconds and it releases resource R_{i+1} which is acquired by the requesting transaction T_i . It has been allotted the resource for the t seconds which is the time for which T_{i+1} has been suspended. T_i is supposed to utilize R_{i+1} and execute successfully in t seconds. If T_i successfully executes before t seconds it sends a message to coordinator that it has successfully executed and to resume transaction T_{i+1} and gives its resource R_{i+1} back to T_{i+1} . If T_i is not able to complete its execution within t second coordinator preempts resource R_{i+1} from T_i and provides it back

to T_{i+1} . The value R_{i+1} is the value partially updated by T_i . Now T_{i+1} will check whether T_i is still requesting for R_{i+1} . If it is requesting, T_{i+1} informs coordinator and is suspended again for some random t seconds and resource R_{i+1} is again allotted to T_i , T_i acquires it and resumes its execution and when completed before t seconds T_i informs coordinator to resume T_{i+1} and gives back resource R_{i+1} to T_{i+1} . Similarly coordinator blocks T_n for some random t seconds and it releases resource R_n which is acquired by the requesting transaction T_{n-1} . It has been allotted the resource for the t seconds which is the time for which T_n has been suspended. T_{n-1} is supposed to utilize R_n and execute successfully in t seconds. If T_{n-1} successfully executes before t seconds it sends a message to coordinator that it has successfully executed and to resume transaction T_n and gives its resource R_n back to T_n . If T_{n-1} is not able to complete its execution within t seconds coordinator preempts resource R_n from T_{n-1} and provides it back to T_n . The value of R_n is the value partially updated by T_{n-1} . Now T_n checks whether T_{n-1} is still requesting for R_n . If it is requesting T_n informs coordinator and is suspended again for some random t seconds and resource R_n is again allotted to T_{n-1} , T_{n-1} acquires it and resumes its execution and when completed before t seconds T_{n-1} informs coordinator to resume T_n and gives back resource R_n to T_n .

LITERATURE SURVEY:

Dinning Philospher Problem:

Research Paper: Part of the [Lecture Notes in Computer Science](#) book series (LNTCS, volume 4228).

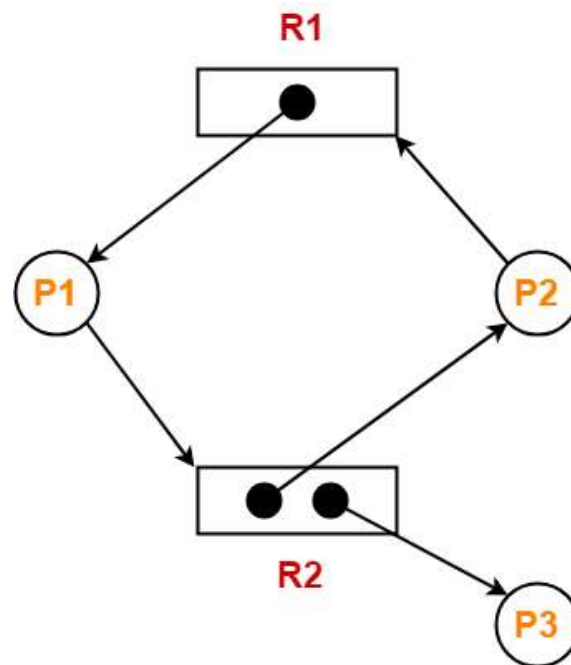
Author: [Jürg Gutknecht](#).

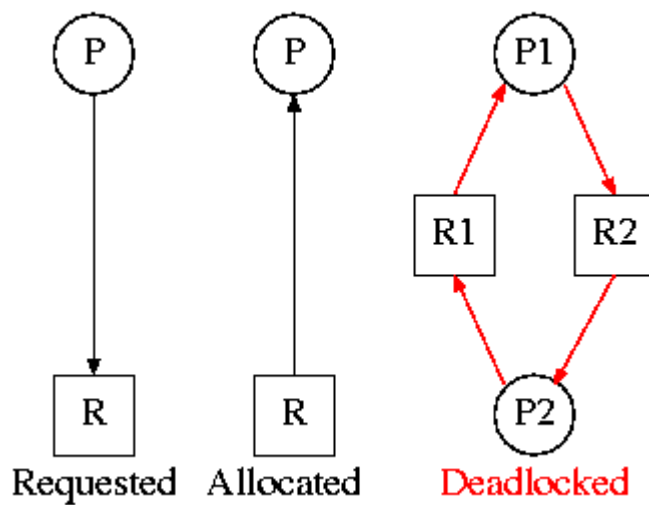
Dead Lock Resolution Techniques:

Journal Refence: International Journal of Scientific and Research Publications,
Volume 3, Issue 7, July 2013.

Authors: Pooja Chahar, Surjeet Dalal

PROPOSED ARCHITECTURE / Algorithm:





RESOLUTIONALGORITHMS:

Here we will discuss two algorithms for deadlock resolution which uses different criterion for victim selection

A. Resolution by using Timestamp

One of the most commonly used technique for deadlock resolution is timestamp based approach for selecting the victim. In this approach, a timestamp is allocated to each process as soon as it enters the system. The timestamp of the younger process is greater than the timestamp of older process. According to this approach, the victim is selected on this timestamps, the process with the higher timestamp is aborted, that is the youngest process is selected as the victim and is aborted in order to break the deadlock cycle. The goal behind choosing the youngest process as victim is that the youngest process would have used less resources and less CPU time as compared to older process. One problem with this technique is that it can cause starvation problem because every time a younger process is aborted which can starve the younger process from completion.

B. Time Efficient Deadlock Resolution Algorithm Deadlock:

is a major concern in a distributed system, since resources are shared among processes at sites distributed across a network. One of the most accepted methods of deadlock handling is detection and resolution. Both deadlock prevention and avoidance strategies are conservative solutions, whereas deadlock detection is optimistic. In deadlock detection and resolution, deadlocks are allowed to occur. Periodically, or on certain conditions, a detection algorithm is executed; if any deadlock state is found, resolution is undertaken. To resolve a detected deadlock, the system must abort one or more processes involved in the deadlock and release the resources allocated to the aborted processes. Here deadlock resolution with reusable resources is considered. In resolving a deadlock state, it is desirable to minimize the number of processes to abort to make the system deadlock-free. Concept of release set is introduced here. A release set is a set of one or more processes that can be reduced if a process is aborted and its resources are released. The release set is represented by $R(p_i)$. For example release set of process P7 and P5 in is $R(P7) = \{P8\}$ and $R(P5) = \{P6, P7, P8\}$.

IMPLEMENTATION DETAILS (Coding):

Approach Using Mutex Locks and P-Threads:

Code:

```
#include<stdio.h>
#include<string.h>
#include<pthread.h>
#include<stdlib.h>
#include<unistd.h>

// Definitions for number of philosophers,
// left philosopher and right philosopher
```

```

#define NUMPHIL 5
#define LEFT ((id-1)+NUMPHIL) % 5
#define RIGHT (id+1) % 5

//Array for number of philosophers and Enum for their states
enum{ THINKING, HUNGRY, EATING } state[NUMPHIL];
int identity[5] = {0,1,2,3,4};

// Mutex, condition variable array for each philosopher,
// and the phillosphers threads array
pthread_mutex_t lock;
pthread_cond_t cond[NUMPHIL];
pthread_t phil[NUMPHIL];

// Funtion declarations
void think(int id);
void pickup_forks(int id);
void eat(int id);
void return_forks(int id);
void *philosopher(void *num);

int main(){
    //Initialize the mutex lock and conditional variable
    int i=0;

    if (pthread_mutex_init(&lock,NULL) != 0) {
        printf("\n mutex init has failed\n");
        return 1;
    }

    //Create the five philosophers threads
    for (i = 0; i < NUMPHIL; i++) {

```

```

        if(pthread_cond_init(&cond[i], NULL)!=0){
            printf("\n cond init has failed\n");
            return 1;
        }
    }

//Create the five philosophers threads
for (i = 0; i < NUMPHIL; i++) {
    pthread_create(&phil[i], NULL, philosopher, &identity[i]);
}

//Join all five philosophers threads
for (i = 0; i < NUMPHIL; i++) {
    pthread_join(phil[i], NULL);
}

//Destroy the mutex and conditional variables
pthread_mutex_destroy(&lock);
for (i = 0; i < NUMPHIL; i++) {
    pthread_cond_destroy(&cond[i]);
}
return 0;
}

//Philosopher wastes time to think now
void think(int id){
    int thinkTime = ((rand()) % 3) +1;

    printf("Philosopher %d is thinking for %d seconds\n", id, thinkTime);
    sleep(thinkTime);
    printf("Philosopher %d reappears from sleep from thinking\n", id);
}

```

```

//Checks to see if the philopher can pick up thier adjacent forks to eat
//If succesful, change enum state to HUNGRY FIRST AND EATING AFTER
//If not, philosopher is blocked by their conditional variable cond[id]
void pickup_forks(int id){
    int left = LEFT;
    int right = RIGHT;
    pthread_mutex_lock(&lock);
    state[id] = HUNGRY;

    while((state[id] == HUNGRY) && ((state[left] == EATING) || (state[right]
== EATING)))){
        printf("Philosopher %i is hungry and waiting to pickup forks to eat
\n", id);
        pthread_cond_wait(&cond[id], &lock);
    }
    state[id] = EATING;
    printf("Philosopher %d is allowed to eat now \n", id);
    pthread_mutex_unlock(&lock);
}

//After getting the forks, the philosopher can eat now
void eat(int id){
    int eatingTime = ((rand()) % 3) +1;

    printf("Philosopher %d is eating for %d seconds\n", id, eatingTime);
    sleep(eatingTime);
    printf("Philosopher %d reappears from sleep from eating\n", id);
}

//After the philosophers eat, put down the forks and change enum state to
THINKING.

```

```

//Signal the current left and right phillosphers blocked by their conditional
//variable to have a chance to see if they can eat now
void return_forks(int id){
    int left = LEFT;
    int right = RIGHT;
    pthread_mutex_lock(&lock);
    state[id]= THINKING;

    printf("Philosopher %d has put down forks\n", id);
    pthread_cond_signal(&cond[left]);
    printf("Philosopher %d signaled philosopher %d to see if it can eat\n", id,
left);
    pthread_cond_signal(&cond[right]);
    printf("Philosopher %d signaled philosopher %d to see if it can eat\n", id,
right);
    pthread_mutex_unlock(&lock);
}

//Main philosopher method  creating the five phillospher threads
void* philosopher(void* num){
    int id = *((int *) num);

    while(1){
        think(id);
        pickup_forks(id);
        eat(id);
        return_forks(id);
    }
}

```

RESULT SET:


```

C:\Users\HP\Documents\OS\Shanmugas
Philosopher 0 is thinking for 3 seconds
Philosopher 2 is thinking for 3 seconds
Philosopher 1 is thinking for 3 seconds
Philosopher 3 is thinking for 3 seconds
Philosopher 4 is thinking for 3 seconds
Philosopher 1 reappears from sleep from thinking
Philosopher 4 reappears from sleep from thinking
Philosopher 2 reappears from sleep from thinking
Philosopher 0 reappears from sleep from thinking
Philosopher 3 reappears from sleep from thinking
Philosopher 1 is allowed to eat now
Philosopher 1 is eating for 3 seconds
Philosopher 4 is allowed to eat now
Philosopher 4 is eating for 3 seconds
Philosopher 2 is hungry and waiting to pickup forks to eat
Philosopher 0 is hungry and waiting to pickup forks to eat
Philosopher 3 is hungry and waiting to pickup forks to eat
Philosopher 1 reappears from sleep from eating
Philosopher 1 has put down forks
Philosopher 4 reappears from sleep from eating
Philosopher 1 signaled philosopher 0 to see if it can eat
Philosopher 1 signaled philosopher 2 to see if it can eat
Philosopher 1 is thinking for 2 seconds
Philosopher 0 is hungry and waiting to pickup forks to eat
Philosopher 4 has put down forks
Philosopher 4 signaled philosopher 3 to see if it can eat
Philosopher 4 signaled philosopher 0 to see if it can eat
Philosopher 4 is thinking for 2 seconds
Philosopher 2 is allowed to eat now
Philosopher 2 is eating for 3 seconds
Philosopher 3 is hungry and waiting to pickup forks to eat
Philosopher 0 is allowed to eat now
Philosopher 0 is eating for 3 seconds

```

APPROACH USING SEMAPHORES:

CODE:

```

#include <pthread.h>
#include <semaphore.h>
#include <stdio.h>

#define N 5
#define THINKING 2
#define HUNGRY 1
#define EATING 0
#define LEFT (phnum + 4) % N
#define RIGHT (phnum + 1) % N

int state[N];
int phil[N] = { 0, 1, 2, 3, 4 };

```

```

sem_t mutex;
sem_t S[N];

void test(int phnum)
{
    if (state[phnum] == HUNGRY
        && state[LEFT] != EATING
        && state[RIGHT] != EATING) {
        // state that eating
        state[phnum] = EATING;

        sleep(2);

        printf("Philosopher %d takes fork %d and %d\n",
               phnum + 1, LEFT + 1, phnum + 1);

        printf("Philosopher %d is Eating\n", phnum + 1);

        // sem_post(&S[phnum]) has no effect
        // during takefork
        // used to wake up hungry philosophers
        // during putfork
        sem_post(&S[phnum]);
    }
}

// take up chopsticks
void take_fork(int phnum)
{
    sem_wait(&mutex);

```

```
// state that hungry
state[phnum] = HUNGRY;

printf("Philosopher %d is Hungry\n", phnum + 1);

// eat if neighbours are not eating
test(phnum);

sem_post(&mutex);

// if unable to eat wait to be signalled
sem_wait(&S[phnum]);

sleep(1);
}

// put down chopsticks
void put_fork(int phnum)
{

    sem_wait(&mutex);

    // state that thinking
    state[phnum] = THINKING;

    printf("Philosopher %d putting fork %d and %d down\n",
           phnum + 1, LEFT + 1, phnum + 1);
    printf("Philosopher %d is thinking\n", phnum + 1);

    test(LEFT);
    test(RIGHT);
```

```
    sem_post(&mutex);
}

void* philosopher(void* num)
{
    while (1) {

        int* i = num;

        sleep(1);

        take_fork(*i);

        sleep(0);

        put_fork(*i);
    }
}

int main()
{
    int i;
    pthread_t thread_id[N];

    // initialize the semaphores
    sem_init(&mutex, 0, 1);

    for (i = 0; i < N; i++)

        sem_init(&S[i], 0, 0);
```

```

for (i = 0; i < N; i++) {

    // create philosopher processes
    pthread_create(&thread_id[i], NULL,
                  philosopher, &phil[i]);

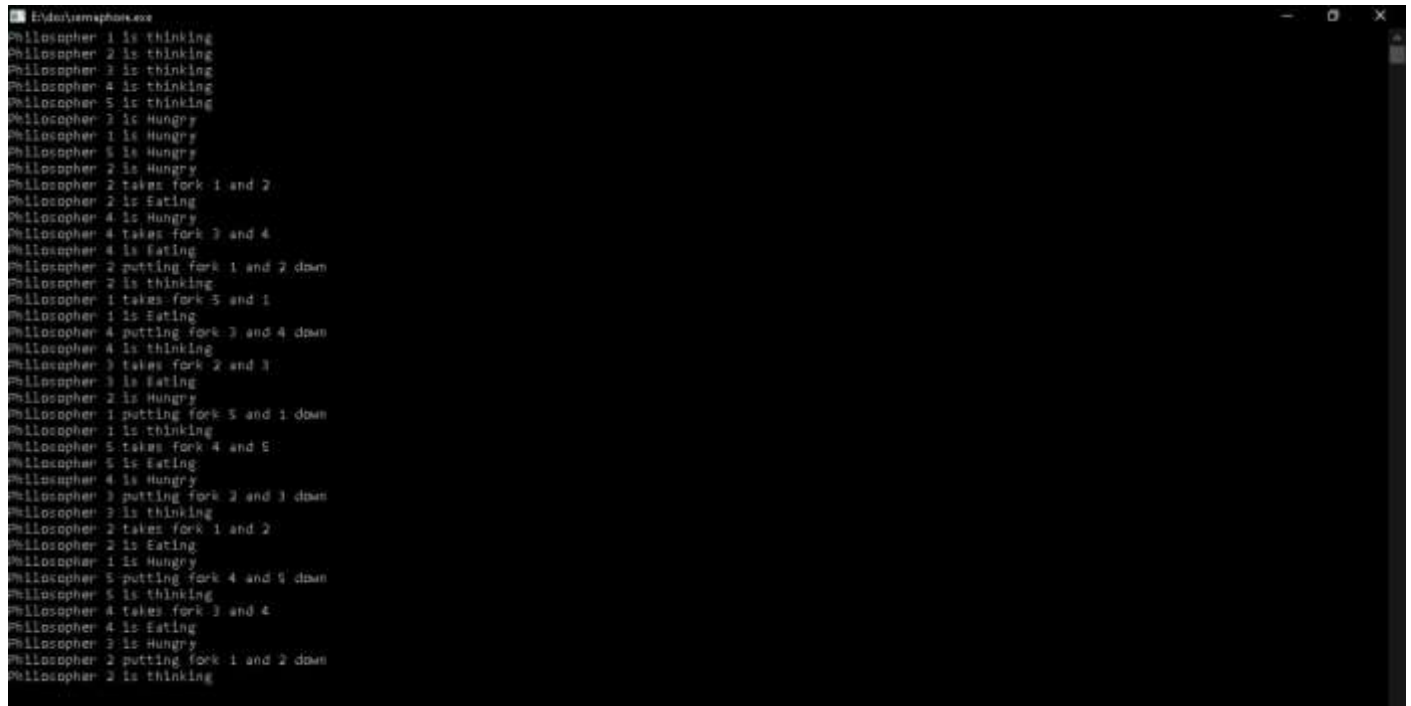
    printf("Philosopher %d is thinking\n", i + 1);
}

for (i = 0; i < N; i++)

    pthread_join(thread_id[i], NULL);
}

```

Result Set:



```

E:\dev\semaphores.exe
Philosopher 1 is thinking
Philosopher 2 is thinking
Philosopher 3 is thinking
Philosopher 4 is thinking
Philosopher 5 is thinking
Philosopher 3 is Hungry
Philosopher 1 is Hungry
Philosopher 5 is Hungry
Philosopher 2 is Hungry
Philosopher 2 takes fork 1 and 2
Philosopher 2 is Eating
Philosopher 4 is Hungry
Philosopher 4 takes fork 3 and 4
Philosopher 4 is Eating
Philosopher 2 putting fork 1 and 2 down
Philosopher 2 is thinking
Philosopher 1 takes fork 5 and 1
Philosopher 1 is Eating
Philosopher 4 putting fork 3 and 4 down
Philosopher 4 is thinking
Philosopher 3 takes fork 2 and 3
Philosopher 3 is Eating
Philosopher 2 is Hungry
Philosopher 1 putting fork 5 and 1 down
Philosopher 1 is thinking
Philosopher 5 takes fork 4 and 5
Philosopher 5 is Eating
Philosopher 4 is Hungry
Philosopher 3 putting fork 2 and 3 down
Philosopher 3 is thinking
Philosopher 2 takes fork 1 and 2
Philosopher 2 is Eating
Philosopher 1 is Hungry
Philosopher 5 putting fork 4 and 5 down
Philosopher 5 is thinking
Philosopher 4 takes fork 3 and 4
Philosopher 4 is Eating
Philosopher 3 is Hungry
Philosopher 2 putting fork 1 and 2 down
Philosopher 2 is thinking

```

PERFORMANCE EVALUATION:

The results that we get with our approach are quite very good we resolved the deadlock occurrence with our minimal code instances.

INFERENCE:

Our Observation and overview regarding problem statement we took and the final result what we achieve with our code is an instance to the real time to understand the Locks, Threads and Dead lock Occurrence with solution has been observed and implemented successfully.

SCOPE FOR IMPROVEMENT:

We approached with all the existing approaches which have minimal time required as in growing technology we have many advance tools so there is a scope of improvement in the code and algorithm. We can figure out.

CONCLUSION:

We achieved a efficient approach to solve the dining philosopher problem with two approaches. Deadlock is a major problem in operating systems. However there are several 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.

References:

- [1]. Yan Cai, k.Zhai, Shngru wu, W.k. Chan,” Synchronizing Threads Globally to Detect Real Deadlocks For Multithreaded Programs”, ACM 2013
- [2]. Jeremy D.Buhler, Kunal Agrawal, Peng Li, Roger D. Chamberlain, “Efficient Deadlock Avoidance For Streaming Computation With Filtering”, 2012 ACM, February 2012.
- [3]. Kunwar Singh Vaisla, Menka Goswami, Ajit Singh, “VGS Algorithm - an Efficient Deadlock Resolution Method”, International Journal of Computer Applications (0975 – 8887), Vol.44- No. 1, April 2012.
- [4]. Selvaraj Srinivasan, R. Rajaram, “A decenreralized deadlock detection and resolution algorithm for generalized model in distributed systems”, january 2011.
- [5]. Selvaraj Srinivasan, R. Rajaram, “A decenreralized deadlock detection and resolution algorithm for generalized model in distributed systems”, January 2011.
- [6]. Iryna Felko, TU Dortmund, “Simulation based Deadlock Avoidance and Optimization in Bidirectional AGVS”, 2011 ACM, march 2011.
- [7]. Prodromos Gerakios, Nikolaso Papaspyrou, Konstantinos, Vekris,”Dynamic Deadlock Avoidance in System Code Using Statically Inferred Effects”, 2011 ACM, October 2011.
- [8]. Peng Li, kunal agrawal, JeremyBuhler, Roger D. Chamberlain,”Deadlock Avoidance for Streaming Computations with Filtering”, 2010 ACM, June 2010.
- [9]. Srinivasan Selvaraj, R. Ramasamy, “An Efficient Detection and Resolution of Generalised Deadlocks in Distributed Systems”, International Journal of Computer Applications, vol 1- no. 19
- [10].Hari K. Pyla, Srinidhi Varadarajan, “Avoiding Deadlock Avoidance”, 2010 ACM, September 2010.
- [11].Pallavi Joshi, Mayur Naik, “A randomized Dynamic Program Analysis Technique for Detecting Read Deadlocks”, 2009 ACM, June 2009.

- [12].Yibei Ling, Shigang Chen and Cho-Yu Jason Chiang, “On Optimal Deadlock Detection Scheduling”, IEEE Transactions On Computers, Vol. 55, No. 9, September 2006.
- [13].Lee, S., “Fast, Centralized Detection and Resolution of Distributed Deadlocks in the Generalized Model”, IEEE Trans. On Software Engineering, Vol. 30, NO. 9, 561-573, 2004
- [14].D. Manivannan and Mukesh Singhal, “An Efficient Distributed Algorithm for Detection of Knots and Cycles in a Distributed Graph”, IEEE Transactions on Parallel And Distributed Systems, Vol. 14, No. 10, October 2003.
- [15].Terekhov, T. Camp, “Time efficient deadlock resolution algorithms”, June 1998.

THANK YOU