Deadlock Prevention Algorithm in Distributed Systems,
Spring 2013

Submitted to:             Submitted by:
Dr. Wang                 Rohit Varma
                          Rohit Dhawan
                          Nitish Bagul
Abstract:
This project presents a software solution to deadlocks in terms of deadlock detection. This new method of preventing deadlocks in resource sharing for distributed system is based on the notion of coloring the nodes of waitfor graph and is built on a signalling mechanism which can be implemented on an underlying routing protocol.
2. Introduction:

2.1 Objective:
The main objective of this project is to implement deadlock prevention algorithm in resource sharing for distributed systems. The traditional algorithms like Banker’s Algorithm have many disadvantages. Also when it comes to distributed systems deadlock is much more severe and complex problem for which Robust and Full-proof algorithm is needed.

Our algorithm addressed the problem with its novel approach which is based on the notion of coloring the nodes of the waitfor graph, and is built on a signalling mechanism which can be implemented on an underlying routing protocol. The algorithm supports multiple resources and multiple outstanding requests.

2.2 Problem Addressed:
In a Distributed Computing System, Autonomous processes are linked by a network and do not have any global memory, but communicate through messages. These processes use resources or share information local to them or available over the network. The processes may request for resources in any order which is not known apriori. The requested resources may be available or locked by other processes. The set of the running processes might request for the same resources and no single process can start executing as each process is waiting on another process which is a part of this cycle causing a deadlock.

2.3 Relation of the Project to Operating Systems:
Deadlock is a potential problem in any operating system. Deadlocks in distributed systems are similar to deadlocks in centralized systems. In centralized systems, we have one operating system that can oversee resource allocation and know whether deadlocks are (or will be) present. With distributed processes and resources it becomes harder to detect, avoid, and prevent deadlocks.

2.4 Disadvantages of other approaches:
- The problem that arises in the case of deadlock prevention algorithms with a centralized memory systems is that if the central site fails then the entire system breaks down.
- In many of the deadlock detection and prevention algorithms, some deadlocks are never detected which defeats the purpose of such an algorithm, while some algorithms detect false deadlocks thus increasing the number of rollbacks, and some other algorithms adopt an over-cautious approach in handling the resource requests leading to unnecessary rollbacks.

2.5 Reason why our approach is better:
- The amount of unnecessary rollback is significantly less compared to existing algorithms.
• The overall message complexity of our algorithm is \( O(n) \). If a process makes a sequence of calls as activate, ack[activate], request, ack[request], grant, release, ack[release], the constant factor of the message complexity (\( O(n) \)) can be high (4 to 6).

2.6 Statement of the problem

Resource sharing in distributed systems is a complex issue. Invoking Deadlock detection algorithm for every resource allocation increases computational cost and complexity. This phenomenon of resource allocation is much more complex when it comes to Distributed Systems and hence the chances of Deadlock are pretty high. The solution proposed here, will act as a guideline for design and development of a robust distributed system which will prevent deadlocks significantly and efficiently.

2.7 Area or Scope of investigation

The efficient way of handling deadlocks may differ according to the design of the system. When it is the case of centralised shared memory system, deadlock prevention can be a useful technique. The proposed algorithm if implemented in accordance with the hardware can greatly improve the performance of the distributed system.

3. Theoretical bases and literature review:

3.1 Definition of the Problem
Deadlock is one of the major problems in the field of distributed systems as the set of the running processes might request for the same resources and no single process can start executing as each process is waiting on another process which is a part of this cycle.

For example, if we have 3 computers 1, 2, and 3, with resources respectively, A, B, and C and D and we have three transactions T1, T2, and T3 that execute as indicate below:

<table>
<thead>
<tr>
<th>TRANSACTION T1</th>
<th>TRANSACTION T2</th>
<th>TRANSACTION T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Transfer from B to A and D)</td>
<td>(Transfer from C to B)</td>
<td>(Transfer from A to C)</td>
</tr>
</tbody>
</table>

```
Lock(D)  
Deposit to D

Lock(A)  
Deposit to A

Lock(B)  
Deposit to B

Lock(C)  
Deposit to C

Lock(C)  
Deposit to C

Lock(A)  
Deposit to C

Lock(A)  
Deposit to C

Withdraw from B  
Withdraw from C  
Withdraw from A

Unlock A, B, D  
Unlock B, C  
Unlock A, C
```

We have the wait for graph

![Wait for Graph](image)

it has a loop, thus we have a deadlock.

### 3.2 Theoretical background of the problem

Designing a distributed system is a complex phenomenon as it consists of set of autonomous processes linked by a network. Considering that each machine gets a partial view of the global system, it adds...
some extra complexity to the already cumbersome task of designing the system. Handling deadlocks in such a system can be very difficult without an accurate and generic approach. This project aims to aid the designers/developers to design a system which will be potentially free from deadlocks.

3.3 Related research to solve the problem
In past many approaches and attempts have been made to address deadlocks. Starting with bankers algorithm, which has the complexity of $O(n^2m)$, many of those tried to address the issue with various algorithms having different complexities. Lot of research work is carried out in the field of deadlock prevention with a centralized shared memory system. The main problem with a centralized approach is vulnerability to failure at the single central site. Obermarck has come up with an algorithm in decentralized systems for fault recovery. Also many algorithms have been proposed for the distributed systems in terms of Deadlock detection, some of prominent ones include Goldman, Isloor-Marsland, Haas-Mohan. Similar research was conducted by Ajay Datta, Ramesh Javagal and Sukumar Ghosh where they proposed the algorithm for distributed systems in terms of deadlock prevention. This is the basis of the algorithm proposed in this paper.

3.4 Solving this problem
As we are dealing with distributed system, the deadlock model presented is AND model of deadlock which is more general than one-resource model. It belongs to the class of age-chasing algorithms where the signals are propagated along the edges of the Wait-for graph. By using Wait-For Graph and manipulating it with the notion of coloring the nodes, a signalling mechanism can be used on which the graph is built.

3.5 How our solution differs from others
- Even though there have been lot of solutions to deadlock in distributed systems in terms of deadlock avoidance and detection, No significant attempts have been made in case deadlock prevention.
- Our approach uses the notion of Wait-For Graph built on a signalling mechanism.
- Our algorithm supports multiple resources and multiple outstanding requests.

3.6 Why our solution is better
- By coloring the nodes this algorithm avoids unnecessary rollbacks which is a serious issue in other overcautious approaches.
- Edge Chasing algorithm when used in distributed system provides distinct advantage, when used for distributed system.
4. Hypothesis:

4.1 Positive Hypothesis
Our approach with use of wait for graph built on signalling system improves the overall system performance by efficiently preventing deadlocks. It also eliminates unnecessary rollbacks to greater extent with its overcautious approach.

4.2 Multiple Hypothesis

- This algorithm supports multiple resources and multiple outstanding requests.
- A process is permitted to request a set of resources.
- A process cannot proceed with its execution until all the requested resources are granted.
- A resource can be released only by the process holding it.

5. Methodology:

5.1 Collection/Generation of input data

- Distributed system composed of a set of individual machines \( \{D_1, D_2, D_3, \ldots, D_n\} \)
- System Matrix M with size \((m \times n)\) where ‘m’ is number of resources and ‘n’ is number of process for the resource allocation in wait for graph.

5.2 Solving the problem

5.2.1 Algorithm design
Procedure send_request;
{process $P_j$ sending a request to $P_k$; process $P_j$ wants a resource owned by $P_k$}

begin

    send activate to every node in the WFG;

    wait for ack[activate] from every node;

    send request for a resource to $P_k$ and wait for grant, ack or nack;

    if grant is received then send deactivate signal to every node;

    if ack is received then

        begin

            send deactivate to every node;

            insert $P_k$ in $P_j$’s dependent-set;

            send get color; {change color since the successor is changed}

            wait for ackcolor;

            wait for the resource;

        end

    else if nack is received then

        begin

            send deactivate to every node;

            rollback;

        end
Procedure receive_request;
{process pi receives a request for a resource from pi; Pk is the owner of the resource}

begin

    if pi = pk then {pi is the owner of the resource}

        if resource is free then

            send grant signal

        else if resource-queue not empty then

            begin

                forward request to tail(resource-queue);

                exit;

            end;

        send get color;

    wait for ackcolor1;

    own color = max(color of processes in dependent-set);

    if own color = request color then send nack

else
if inactive then send ack

else if own color < request color then

    begin

        enqueue pj in Pk’S resource-queue;

        insert (pj,pk) in pi’s wait-sel; {add an edge
        (pi, pi) in the WFG}

        send acL;

        end

    else send nack.;

end; { receive-request}

Procedure receive_grant;
{process pj receives a grant signal for a resource from Pk 1}

begin

    delete Pk from pj’s dependent-set;

    send get color; {change color since the successor is changed}

    wait for ackcolor1;

    if dependent-set empty
    then execute
    else wait for other resources;

end; {receive-grant}

Procedure send_release;
\{process pj releases a resource owned by Pk\}

begin

\hspace{1cm} send release to Pk;

\hspace{1cm} wait for ack\{release\};

\hspace{1cm} if 3 pi I (pi&) E pj’s wait-set then

\hspace{2cm} remove \{pi ,PE\} from wait-set; \{remove the edge(pi,pj) from the WFG\}

end; \{send-release\}

**Procedure receive_release;**

\{process P receives a release signal from pj; Pk is the owner\}

begin

\hspace{1cm} dequeue pj from resource-queue;

\hspace{1cm} send ack\{release\} to pj;

\hspace{1cm} if resource-queue not empty then

\hspace{2cm} send grant to head(resource-queue);

end; \{ receive-release\}

**Procedure rollback;**

\{process pj has to rollback\}

begin
for every process \( P_k \) in \( pj \)'s dependent-set

    call send-release; \{remove all requests\}

for every owner \( pk \) of the resources held by \( pj \)

    call send-release; \{release all resources\}

end; \{rollback\}

**5.2.2 Language used**

We are using Java to test the proposed solution.

**5.3 Generating output**

Input will be in the form of files containing the data as mentioned above in the input section. The output will be in terms of simulation of the presented algorithm. Precisely it would be in the form of resource allocation summary with number of cycles.

**5.4 Test against hypothesis**

The results from the proposed algorithm will be compared to various approaches. This performance evaluation will help us prove the hypothesis true.

**6. Implementation**

**Design and Flowchart:**

The following two flowcharts represent the main functionality of our program
a. Send Request (Occurs when a process sends a request for resource from another process)

b. Receive Request (Occurs when process receives a request from another process for a resource)
Start

Process P1 requests a Resource owned by P2

Check if resource is available

Yes

Grant is received

No

If Resource is unavailable, ACK or NACK can be sent

NACK

NACK received

ACK

ACK Received

Update P1's dependent set, Change color of node

Stop
7. Data analysis and Discussion

7.1 Output generation
We considered a Distributed system with 4 processes namely P1, P2 and P3 and four Resources in the system R1, R2, R3 and R4. In our algorithm we have considered that only one resource can be owned by a process at any time.

The following inputs were given:

a. P1 sends a request to P4 for resource R2.
b. P2 sends a request to P4 for resource R2
c. P3 sends a request to P4 for resource R2
d. P1 sends a request to P3 for resource R3 owned by it.
   // P1 Rollback

The diagram below illustrates the working

Figure 2. Example of a process rollback. (a) process p1 makes a request for a resource to process p3. (b) process p1 rolls back.
Output:

P1 sends Request to P4 for R4
Resource is available
Grant Received
Resource Queue of P4 = {P1}
Color of P1 = 4

P2 sends Request to P4 for R4
Resource is busy
ACK received
Color of P2 = 4
Resource Queue of P4 = {P1,P2}
Wait set of P4 = {(P1,P2)}
Dependent Set of P2 = {P4}

P3 sends Request to P4 for R4
Resource is busy
ACK received
Color of P3 = 4
Resource Queue of P4 = {P1,P2,P3}
Wait set of P4 = {(P2,P3)}
Dependent Set of P3 = {P4}

P1 sends Request to P3 for R3
Resource is busy
NACK Received
P1 Rollback

7.2 Output Analysis
In the output above there are four requests made totally

First, P1 sends a request to P4 for resource R4.
The request is granted since the resource is available.
Hence P4 sends a grant to P1.
P1 receives the grant from P4 but waits for other resources it needs for execution.
P1 is added to the resource queue of P4 and its color is changed to its successors color i.e. 4.

Second, P2 sends a request to P4 for resource R4.
The request is not granted since the resource is busy.
P1 sends an ACK to P2.
P2 receives the ACK from P1 waits P1 to complete execution.
P2 is added to the resource queue of P4 and its color is changed to its successors color i.e. 4.
The wait set of P4 is updated with the entry (P1,P2)

Third, P3 sends a request to P4 for resource R4.
The request is not granted since the resource is busy.
P2 (last element in the resource queue) (tail) sends an ACK to P3.
P3 receives the ACK from P2 waits P2 to complete execution.
P3 is added to the resource queue of P4 and its color is changed to its successors color i.e. 4.
The wait set of P4 is updated with the entry (P2, P3)

In the final step
P1 sends a request to P3 for resource R3 which it owns.
The request is not granted since the resource is busy.
P3 sends an NACK to P1.
Hence Process P1 rollback. So P1 releases all its resources and cancels all its requests. After a certain amount of time, P1 can restart.

7.3 Compare Output against Hypothesis
Our approach with use of wait for graph built on signalling system improves the overall system performance by efficiently preventing deadlocks. It also eliminates unnecessary rollbacks to greater extent.

7.4 Abnormal Case Explanation (the most important task)

When we implement threads for the processes to make calls to send request for resources the order of thread is not predictable leading to different outputs each time simulation takes place.
8. Conclusions and Recommendations

8.1 Summary and Conclusions
In this project, we have tried to demonstrate how our algorithm to detect deadlock using WaitForGraph is efficient. Preventing deadlock is crucial to any system as it halts all the processes and resources from performing their tasks. The method implemented in this paper helps in Preventing deadlock in an efficient manner.
By coloring the nodes this algorithm avoids unnecessary rollbacks which is a serious issue in other overcautious approaches.

8.2 Recommendations for future studies
In any deadlock prevention algorithm if the maximum claims of the processes are not known in advance, process rollback is unavoidable. Hence we need to find an optimal solution to limit the number of Rollbacks.
9. Bibliography:


3) An algorithm for Avoiding Deadlock, Md. Nawab Yousuf Ali, Mohammad Zakir Hossain Sarker, East West University, Bangladesh.

4) On Siphon Computation for Deadlock Control in a Class of Petri Nets, ZhiWu Li, MengChu Zhou, IEEE

5) A Novel Deadlock Avoidance Algorithm and Its Hardware Implementation, Jaehwan Lee, Vincent John Mooney III, Georgia Institute of Technology, Georgia, USA.

6) An Algorithm For Preventing Deadlocks In Distributed Systems, Ajoy Kumar Datta, Ramesh Dutt Javagal, Sukumar Ghosh, Department of computer science, University of Nevada, Nevada, USA.
10. **Appendices**

10.1 Program Flowchart

a. Send_Request
b. Receive_Request
Start

Process P1 requests Resource owned by P2

If resource is available

Grant Received

If Resource is unavailable ACK or NACK can be sent

Update P1's dependent set, Change color of node

Stop
Start

Process Pi requests PJ for Resource owned by PK

If Pi==Pk, Pi is owner
  yes -> Grant Received
  no -> Resource Queue is not empty

If Resource Queue is not empty
  yes -> Forward Request to tail of Resource Queue
  no -> if own color == request color

if own color == request color
  yes -> if resource is inactive
  yes -> Send ACK
  no -> Else Send Nack

else -> if own color == request color

else -> enqueue PJ in PKs Resource Queue and insert (PJ, PK) in PKs waitset

Else Send Nack

Stop
10.2 Code:

Class Node:

```java
class Node extends Process {
    public Node(int pn, ArrayList<Resource> rs, int cn) {
        super(pn, rs);
        color_Node = cn;
        // TODO Auto-generated constructor stub
    }

    public static void request_Send(Node n1, Node n2, Resource r) {
        // Send ACK[activate]---receive[activate]

        // check for grant
        Iterator<Resource> ri = n2.resources_owned.iterator();
        // Iterator<Resource> ri2 = n2.current_resources.iterator();

        while (ri.hasNext()) {
            Resource res = ri.next();
            if (res.resource_number == r.resource_number) {
                // if its owned, check for the availability
                if (n2.getAvailable() == 1) {
                    // if available, grant resource
                    System.out.println("Resource Available");
                    System.out.println("Grant Received");
                    n2.setAvailable(0);
                }
            }
        }
    }
}
```
else {
    if (!n2.resource_Queue.isEmpty()) {
        n2.resource_Queue.add(n1);
    }
    if (n2.getColor_Node() == n1.getColor_Node()) {
        System.out.println("Ack Received");
        break;
    } else {
        if (n2.getAvailable() == 1) {
            System.out.println("ACK Received");
        }
        if (n2.getColor_Node() < n1.getColor_Node()) {
            n2.resource_Queue.add(n1);
            // Adding tuple to waitset
            ArrayList<Process_Pair> tuple = new ArrayList<Process_Pair>();
            Process_Pair p = new Process_Pair(n1, n2);
            tuple.add(p);
            n1.setWait_Set(tuple);
        } else {
            System.out.println("NACK Received");
        }
    }
}

} // if_end

} // while_end

} // Request_Send(end)

public void request_Resource() {
}

public void grantResource(Node n1, Node n2, Resource r) {

public class Process {
    public int process_number;
    public ArrayList<Resource> resources_owned;
    
    //To keep track of currently used resource
    public ArrayList<Resource> current_resources;
    
    //To check the owned resource is available
    public static int available=1;

    public ArrayList<Resource> getCurrent_resources() {
        //n2.current_resources.remove(r);
    }
}

Process Class

package process_Implement;

import java.util.ArrayList;
import java.util.Queue;

public class Process {
    public int color_Node;
    
    public int getColor_Node() {
        return color_Node;
    }
    public void setColor_Node(int color_Node) {
        this.color_Node = color_Node;
    }
    public Boolean hasColor(){
        return true;
    }
}
public void setCurrent_resources(ArrayList<Resource> current_resources) {
    this.current_resources = current_resources;
}

public Queue<Process> getResource_Queue() {
    return resource_Queue;
}

public void setResource_Queue(Queue<Process> resource_Queue) {
    this.resource_Queue = resource_Queue;
}

public ArrayList<Process_Pair> getWait_Set() {
    return wait_Set;
}

public void setWait_Set(ArrayList<Process_Pair> wait_Set) {
    this.wait_Set = wait_Set;
}

public ArrayList<Process> getDependent_Set() {
    return dependent_Set;
}

public void setDependent_Set(ArrayList<Process> dependent_Set) {
    this.dependent_Set = dependent_Set;
}

public Queue<Process> resource_Queue;
public ArrayList<Process_Pair> wait_Set;

public ArrayList<Process> dependent_Set;

//Constructor
public Process(int pn, ArrayList<Resource> rs)
{
    process_number = pn;
    resources_owned = rs;
    available = 1;
}

public int getAvailable() {
    return available;
}

public void setAvailable(int available) {
    this.available = available;
}

public int getProcess_number() {
    return process_number;
}

public void setProcess_number(int process_number) {
    this.process_number = process_number;
}

public ArrayList<Resource> getResources_owned() {
    return resources_owned;
}

public void setResources_owned(ArrayList<Resource> resources_owned) {
    this.resources_owned = resources_owned;
}
Class Process_pair

package process_Implement;

public class Process_Pair {

    public Process[] tuple_process = new Process[2];

    public Process[] getTuple_process() {
        return tuple_process;
    }

    public void setTuple_process(Process[] tuple_process) {
        this.tuple_process = tuple_process;
    }

    public Process_Pair(Process p1, Process p2) {
        tuple_process[0] = p1;
        tuple_process[1] = p2;
    }
}

Edge_pair class

package process_Implement;

public class Edges_Pair {

    public Process[] edge_Pair = new Process[2];

    public Edges_Pair(Process p1, Process p2) {
        edge_Pair[0] = p1;
        edge_Pair[1] = p2;
    }
}


**Edge Class**

```java
package process_Implement;

import java.util.ArrayList;
import java.util.Iterator;

public class Edges {

    public ArrayList<Edges_Pair> Edges_WFG;

    public boolean hasEdge(Process p1, Process p2) {
        Iterator<Edges_Pair> it = Edges_WFG.iterator();
        while (it.hasNext()) {
            Edges_Pair obj = it.next();
            if (obj.edge_Pair[0] == p1 && obj.edge_Pair[1] == p2) {
                return true;
            } else {
                return false;
            }
        }
        return false;
    }
}
```

**Main Class for implementation**

```java
package process_Implement;

import java.util.ArrayList;

public class Implementation {

    public static void createEnvironment() {

```
public static void main(String[] args) {
    // Process creation, Allocating resources to processes
    // create resources
    Resource r1 = new Resource(1);
    Resource r2 = new Resource(2);
    Resource r3 = new Resource(3);
    Resource r4 = new Resource(4);

    // Arraylist for resources
    ArrayList<Resource> rs_P1 = new ArrayList<Resource>();
    ArrayList<Resource> rs_P2 = new ArrayList<Resource>();
    ArrayList<Resource> rs_P3 = new ArrayList<Resource>();
    ArrayList<Resource> rs_P4 = new ArrayList<Resource>();

    // Adding elements
    rs_P1.add(r1);
    rs_P2.add(r2);
    rs_P2.add(r2);
    rs_P3.add(r3);
    rs_P3.add(r4);

    // create processes
    Process p1 = new Process(1, rs_P1);
    Process p2 = new Process(2, rs_P2);
    Process p3 = new Process(3, rs_P3);
    Node n1 = new Node(1, rs_P1, 1);
    Node n2 = new Node(2, rs_P1, 2);
    Node n3 = new Node(3, rs_P1, 3);
    Node n4 = new Node(4, rs_P1, 4);

    System.out.println("Process P1 requests P4 for resource R4 ");
    Node.request_Send(n1, n4, r4);
    System.out.println("Process P2 requests P4 for resource R4 ");
    Node.request_Send(n3, n2, r4);
    System.out.println("Process P3 requests P4 for resource R4 ");
    Node.request_Send(n3, n4, r4);
}
System.out.println("Process P1 requests P4 for resource R4 ");
Node.request_Send(n1, n3, r4);

10.3 Input/Output Listing:

Input: Processes P1, P2, P3, P4 located at nodes n1, n2, n3 and n4.

Output:

Number of processes = 4
Number of resources = 4
P1 sends Request to P4 for R4
Resource is available
Grant Received
Resource Queue of P4 = {P1}
Color of P1 = 4

P2 sends Request to P4 for R4
Resource is busy
ACK received
Color of P2 = 4
Resource Queue of P4 = {P1,P2}
Wait set of P4 = {(P1,P2)}
Dependent Set of P2 = {P4}

P3 sends Request to P4 for R4
Resource is busy
ACK received
Color of P3 = 4
Resource Queue of P4 = {P1,P2,P3}
Wait set of P4 = {(P2,P3)}
Dependent Set of P3 = {P4}

P1 sends Request to P3 for R3
Resource is busy
NACK Received
P1 Rollback