RGPVNOTES.IN

Subject Name: **Distributed System**

Subject Code: **IT-6005**

Semester: **6th**

UNIT 2

Distributed Deadlock Detection:

Deadlock:

Deadlock is a fundamental problem in distributed systems. A process may request resources in any order, which may not be known a priori and a process can request resource while holding others. If the sequence of the allocations of resources to the processes is not controlled, deadlocks can occur. A deadlock is a state where a set of processes request resources that are held by other processes in the set.

System Model

Distributed program is composed of a set of n asynchronous processes p1, p2, . . . , pi , . . . , pn that communicates by message passing over the communication network. Without loss of generality we assume that each process is running on a different processor. The processors do not share a common global memory and communicate solely by passing messages over the communication network. There is no physical global clock in the system to which processes have instantaneous access. The communication medium may deliver messages out of order, messages may be lost garbled or duplicated due to timeout and retransmission, processors may fail and communication links may go down.

We make the following assumptions for system models:

- The systems have only reusable resources.
- Processes are allowed to make only exclusive access to resources.
- There is only one copy of each resource.

A process can be in two states: running or blocked. In the running state (also called active state), a process has all the needed resources and is either executing or is ready for execution. In the blocked state, a process is waiting to acquire some resource.

Resource Vs Communication Deadlock:

A deadlock is a condition where a process cannot proceed because it needs to obtain a resource held by other process and itself is holding a resource that the other process needed. We can considered two type of deadlock-

1. **Communication deadlock:**

In communication deadlock processes wait to communicate with other processes. A waiting process can unlock on receiving a communication from any one of these processes.

A set of processes is communication dead-lock if each process in the set is waiting to communication with another process in the set and no process in the set even initiates any further communication, until receives the communication which is waiting (Shown in fig 2.1).

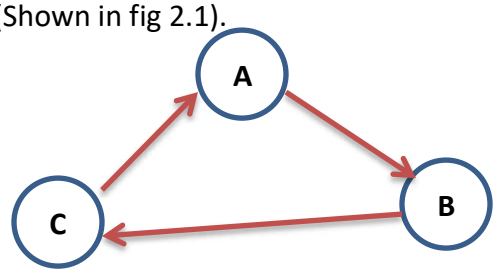


Fig 2.1 Communication deadlock

When a process A is trying to send a message to process B and process B is trying to send a message to process C, which is trying to send a message to process A.

2. **Resource deadlock:**

A deadlock is a condition in a system where a set of processes (or threads) have requests for resources that can never be satisfied. Essentially, a process cannot proceed because it needs to obtain a resource held by another process but it itself is holding a resource that the other process needs. More formally, Coffman defined four conditions have to be met for a deadlock to occur in a system:

a) Mutual exclusion: A resource can be held by at most one process.

b) Hold and wait: Processes that already hold resources can wait for another resource.

c) Non-preemption: A resource, once granted, cannot be taken away.

d) Circular wait: Two or more processes are waiting for resources held by one of the other processes.

A directed graph model used to record the resource allocation state of a system. This state consists of n processes, P1 … Pn, and m resources, R1 … Rm. Shown in fig 2.2:

P1 → R1 means that resource R1 is allocated to process P1.

P1 ← R1 means that resource R1 is requested by process P1.

Deadlock is present when the graph has a directed cycle. An example is shown in Figure 1. Such a graph is called a Wait-For Graph (WFG).

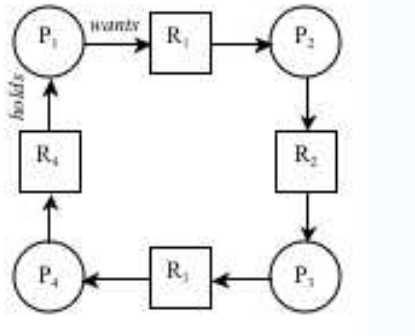


Fig 2.2 Deadlock due to resources

Deadlock Handling Strategy in DS:

Deadlock Prevention:

Deadlock prevention works by preventing one of the four Coffman conditions from occurring.

a) Removing the mutual exclusion condition means that no process will have exclusive access to a resource. This proves impossible for resources that cannot be spooled. But even with spooled resources, deadlock could still occur. Algorithms that avoid mutual exclusion are called non-blocking synchronization algorithms.

b) The hold and wait or resource holding conditions may be removed by requiring processes to request all the resources they will need before starting up (or before embarking upon a particular set of operations). This advance knowledge is frequently difficult to satisfy and, in any case, is an inefficient use of resources. Another way is to require processes to request resources only when it has none. Thus, first they must release all their currently held resources before requesting all the resources they will need from scratch. This too is often impractical. It is so because resources may be allocated and remain unused for long periods. Also, a process requiring a popular resource may have to wait indefinitely; as such a resource may always be allocated to some process, resulting in resource starvation. (These algorithms, such as serializing tokens, are known as the all-or-none algorithms.)

c) The no preemption condition may also be difficult or impossible to avoid as a process has to be able to have a resource for a certain amount of time, or the processing outcome may be inconsistent or thrashing may occur. However, inability to enforce preemption may interfere with a priority algorithm. Preemption of a "locked out" resource generally implies a rollback, and is to be avoided, since it is very costly in overhead. Algorithms that allow preemption include lock-free and wait-free algorithms and optimistic concurrency control. If a process holding some resources and requests for some another resource(s) that cannot be immediately allocated to it, the condition may be removed by releasing all the currently being held resources of that process.

d) The final condition is the circular wait condition. Approaches that avoid circular waits include disabling interrupts during critical sections and using a hierarchy to determine a partial ordering of resources. If no obvious hierarchy exists, even the memory address of resources has been used to determine ordering and resources are requested in the increasing order of the enumeration. Dijkstra's solution can also be used.

Deadlock avoidance

Deadlock avoidance merely works to avoid deadlock. It does not totally prevent it. The basic idea here is to allocate resources only if the resulting global state is a safe state. In other words, unsafe states are avoided, meaning that deadlock is avoided as well.

One famous algorithm for deadlock avoidance in the uniprocessor case is the Banker's Algorithm. Similar algorithms have been attempted for the distributed case.

1. When a process requests a resource, even if it is available, it is not immediately allocated to the process. Instead the systems assumes (pretends) it is so allocated.
2. With this assumption, along with advance knowledge of all the resources needed for all the processes, the system performs some analysis to decide whether or not granting the request is safe.
3. If the request is safe, the resource is granted to the requesting process. Otherwise, the resource is not given to the process at this time.

This type of algorithm requires each site to have access to a global state (requiring too much storage and communication).

Also the checking of the involved data structures (e.g. the Banker's Algorithm tables) must be done in a mutually exclusive fashion.

The Banker's Algorithm:

Allows:

mutual exclusion
wait and hold
no preemption

Prevents:

circular wait

User process may only request one resource at a time. System grants request only if the request will result in a safe state.

An Example

Assume we have the following resources:

5 tape drives
2 graphic displays
4 printers
3 disks

We can create a vector representing our total resources: Total = (5, 2, 4, 3).

Consider we have already allocated these resources among four processes as demonstrated by the following matrix named Allocation.

Process Name	Tape Drives	Graphics	Printers	Disk Drives
Process A	2	0	1	1
Process B	0	1	0	0
Process C	1	0	1	1
Process D	1	1	0	1

The vector representing the allocated resources is the sum of these columns: Allocated = (4, 2, 2, 3).

We also need a matrix to show the number of each resource still needed for each process; we call this matrix Need.

Process Name	Tape Drives	Graphics	Printers	Disk Drives
Process A	1	1	0	0
Process B	0	1	1	2
Process C	3	1	0	0
Process D	0	0	1	0

The vector representing the available resources will be the sum of these columns subtracted from the Allocated vector: Available = (1, 0, 2, 0).

Algorithm:

1) Find a row in the Need matrix which is less than the Available vector. If such a row exists, then the process represented by that row may complete with those additional resources. If no such row exists, eventual deadlock is possible.
2) You want to double check that granting these resources to the process for the chosen row will result in a safe state. Looking ahead, pretend that that process has acquired all its needed resources, executed, terminated, and returned resources to the Available vector. Now the value of the Available vector should be greater than or equal to the value it was previously.
3) Repeat steps 1 and 2 until
   a) all the processes have successfully reached pretended termination (this implies that the initial state was safe); or
   b) Deadlock is reached (this implies the initial state was unsafe).

Following the algorithm sketched above,

- Iteration 1:
  1. Examine the Need matrix. The only row that is less than the Available vector is the one for Process D.
     Need (Process D) = (0, 0, 1, 0) < (1, 0, 2, 0) = Available
  2. If we assume that Process D completes, it will turn over its currently allocated resources, incrementing the Available vector.

     (1, 0, 2, 0) Current value of Available
     + (1, 1, 0, 1) Allocation (Process D)
     ``````````````````
     (2, 1, 2, 1) Updated value of Available

- Iteration 2:
  1. Examine the Need matrix, ignoring the row for Process D. The only row that is less than the Available vector is the one for Process A.
     Need (Process A) = (1, 1, 0, 0) < (2, 1, 2, 1) = Available
  2. If we assume that Process A completes, it will turn over its currently allocated resources, incrementing the Available vector.

     (2, 1, 2, 1) Current value of Available
     + (2, 0, 1, 1) Allocation (Process A)
     ``````````````````
     (4, 1, 3, 2) Updated value of Available

- Iteration 3:
  1. Examine the Need matrix without the row for Process D and Process A. The only row that is less than the Available vector is the one for Process B.
     Need (Process B) = (0, 1, 1, 2) < (4, 1, 3, 2) = Available
  2. If we assume that Process B completes, it will turn over its currently allocated resources, incrementing the Available vector.

     (4, 1, 3, 2) Current value of Available
     + (0, 1, 0, 0) Allocation (Process B)
     ``````````````````
     (4, 2, 3, 2) Updated value of Available

- Iteration 4:
  1. Examine the Need matrix without the rows for Process A, Process B, and Process D. The only row left is the one for Process C, and it is less than the Available vector.
     ``````````````````

Need (Process C) = (3, 1, 0, 0) < (4, 2, 3, 2) = Available

2. If we assume that Process C completes, it will turn over its currently allocated resources, incrementing the Available vector.

(4, 2, 3, 3)	Current value of Available
+ (1, 0, 1, 1)	Allocation (Process C)
(5, 2, 4, 3)	Updated value of Available

Notice that the final value of the Available vector is the same as the original Total vector, showing the total number of all resources:

Total = (5, 2, 4, 2) < (5, 2, 4, 2) = Available

This means that the initial state represented by the Allocation and Need matrices is a safe state.

The safe sequence that assures this safe state is <D, A, B, C>.

Disadvantages of the Banker's Algorithm

- It requires the number of processes to be fixed; no additional processes can start while it is executing.
- It requires that the number of resources remain fixed; no resource may go down for any reason without the possibility of deadlock occurring.
- It allows all requests to be granted in finite time, but one year is a finite amount of time.
- Similarly, all of the processes guarantee that the resources loaned to them will be repaid in a finite amount of time. While this prevents absolute starvation, some pretty hungry processes might develop.
- All processes must know and state their maximum resource need in advance.

Deadlock detection & resolution

Deadlock detection attempts to find and resolve actual deadlocks. These strategies rely on a Wait-For-Graph (WFG) that in some schemes is explicitly built and analyzed for cycles.

In the WFG, the nodes represent processes and the edges represent the blockages or dependencies. Thus, if process A is waiting for a resource held by process B, there is an edge in the WFG from the node for process A to the node for process B.

In the AND model (resource model), a cycle in the graph indicates a deadlock. In the OR model, a cycle may not mean a deadlock since any of a set of requested resources may unblock the process. A knot in the WFG is needed to declare a deadlock. A knot exists when all nodes that can be reached from some node in a directed graph can also reach that node.

In a centralized system, a WFG can be constructed fairly easily. The WFG can be checked for cycles periodically or every time a process is blocked, thus potentially adding a new edge to the WFG. When a cycle is found, a victim is selected and aborted.

Algorithms for detecting distributed deadlock can be handled in three different ways:

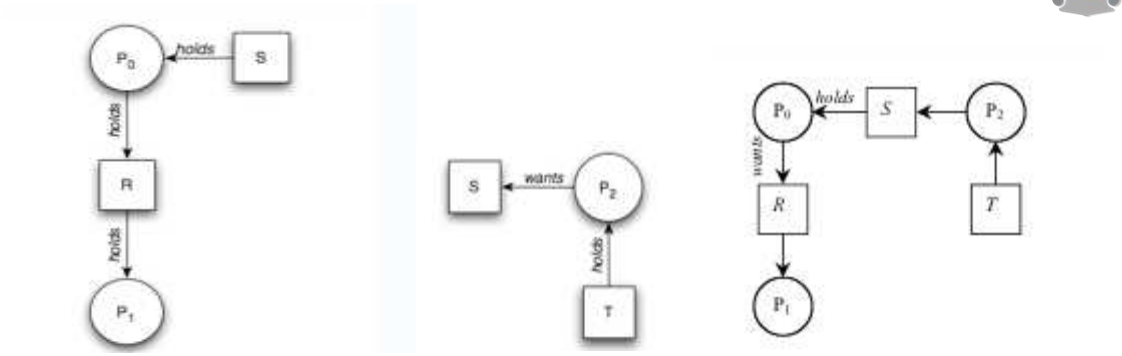
- Centralized
- Distributed
- Hierarchical

Assume that the network supports reliable communication.

Centralized Deadlock Detection

In the distributed case, the individual sub graphs have to be propagated to a central coordinator. A message can be sent each time an arc is added or deleted. If optimization is needed, a list of added or deleted arcs can be sent periodically to reduce the overall number of messages sent.

Here is an example. Suppose machine A has a process P0, which holds the resource S and wants resource R, which is held by P1. The local graph on A is shown in Fig 2.3(a). Another machine, machine B, has a process P2, which is holding resource T and wants resource S. Its local graph is shown in Fig 2.3(b). Both of these machines send their graphs to the central coordinator, which maintains the union (Fig 2.3(c)).



(a) **(b).** **(c)**

Fig 2.3 local graph of each process:

All is well. There is no cycles and hence no deadlock. Now two events occur. Process P1 releases resource R and asks machine B for resource T. Two messages are sent to the coordinator:

message 1 (from machine A): “releasing R”

message 2 (from machine B): “waiting for T”

This should cause no problems (no deadlock). However, if message 2 arrives first, the coordinator would then construct the graph in Figure 2.4 and detect a deadlock. Such a condition is known as false deadlock. A way to fix this is to use Lamport’s algorithm to impose global time ordering on all machines. Alternatively, if the coordinator suspects deadlock, it can send a reliable message to every machine asking whether it has any release messages. Each machine will then respond with either a release message or a negative acknowledgement to acknowledge receipt of the message. If second time again coordinator detect same cycle than it is declared there is a deadlock.

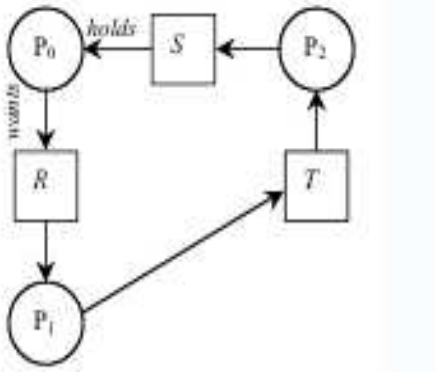


Fig 2.4 Centralized Deadlocks:

Distributed Deadlock Detection

In distributed deadlock detection we have two methods

1. Obermarck's Path-Pushing Algorithm:

Designed for distributed database systems processes are called ``transactions'' T1, T2, … Tn, there is special virtual node Ex transactions are totally ordered

In path-pushing algorithms, distributed deadlocks are detected by maintaining an explicit global WFG. The basic idea is to build a global WFG for each site of the distributed system.

In this class of algorithms, at each site whenever deadlock computation is performed, it sends its local WFG to all the neighboring sites.

After the local data structure of each site is updated, this updated WFG is then passed along to other sites, and the procedure is repeated until some site has a sufficiently complete picture of the global state to announce deadlock or to establish that no deadlocks are present.

This feature of sending around the paths of global WFG has led to the term path-pushing algorithms.

Algorithm:

1) wait for info from previous iteration of Step 3
2) combine received info with local TWFG
   detect all cycles
   break local cycles
3) send each cycle including the node Ex to the external nodes it is waiting for
4) time-saver: only send path to other sites if last transaction is higher in lexical order than the first

Example:

Initially all sites **(**Machine**)** connected like this(shown in fig 2.5):

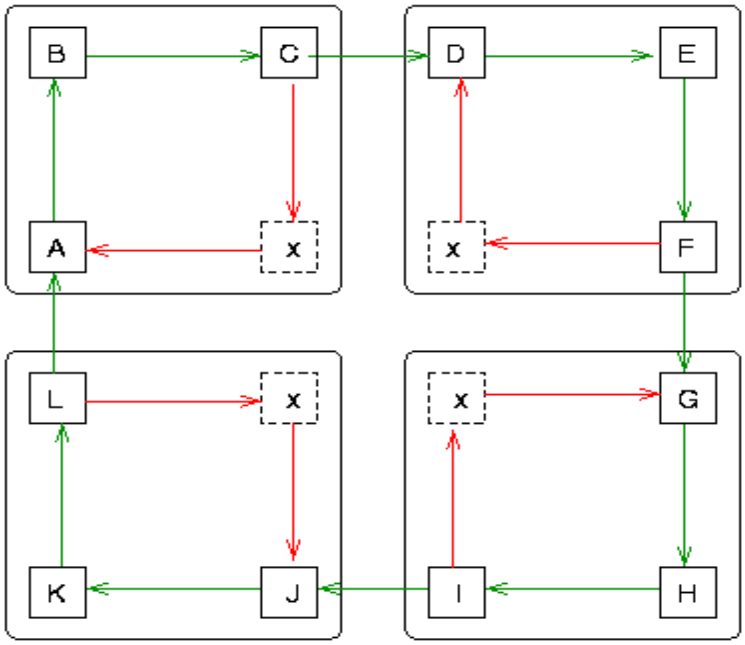


Fig 2.5 Path pushing algorithm initial pass

Iteration 1: After the passing first message (message containing its path (shown in fig 2.6))

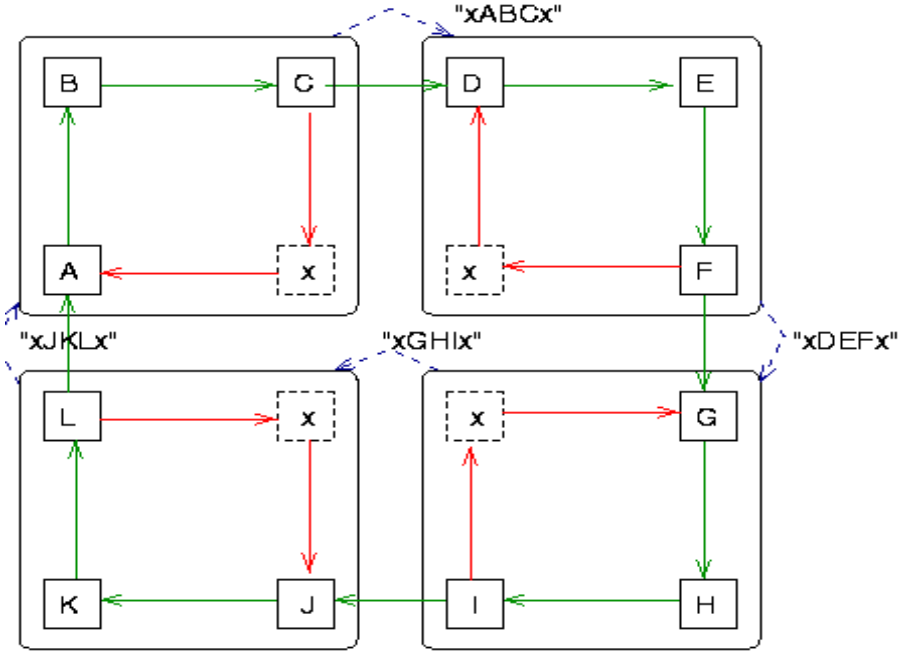


Fig 2.6 Path pushing algorithm first pass

Iteration 2:

In second iteration length of message size will increase shown in fig 2.7:

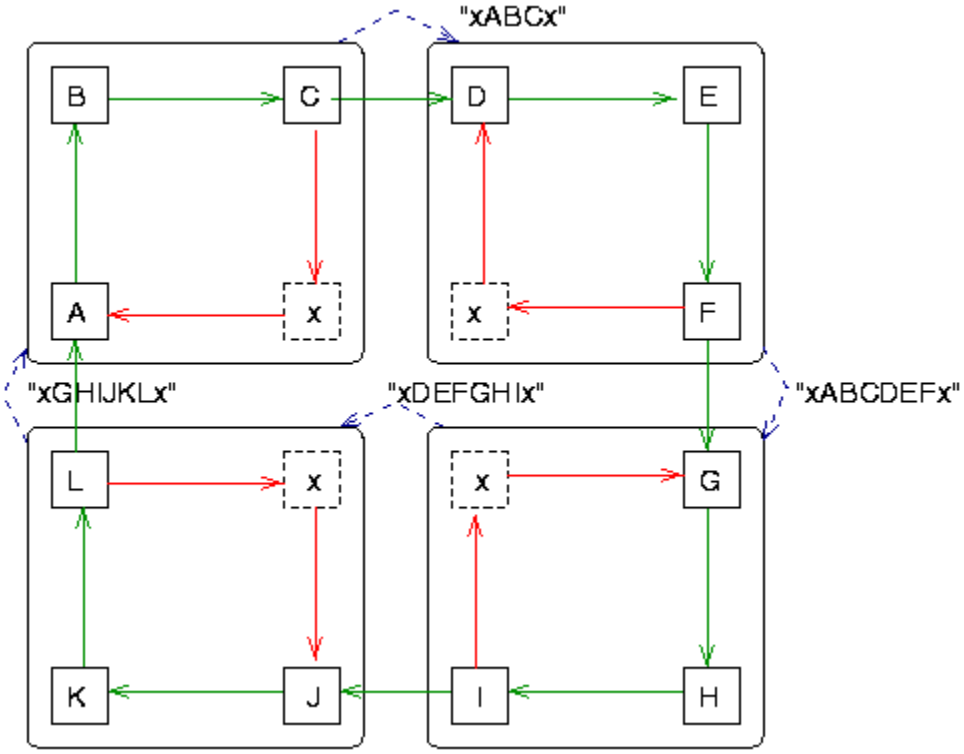


Fig 2.7 Path pushing algorithm second pass

Iteration 3:

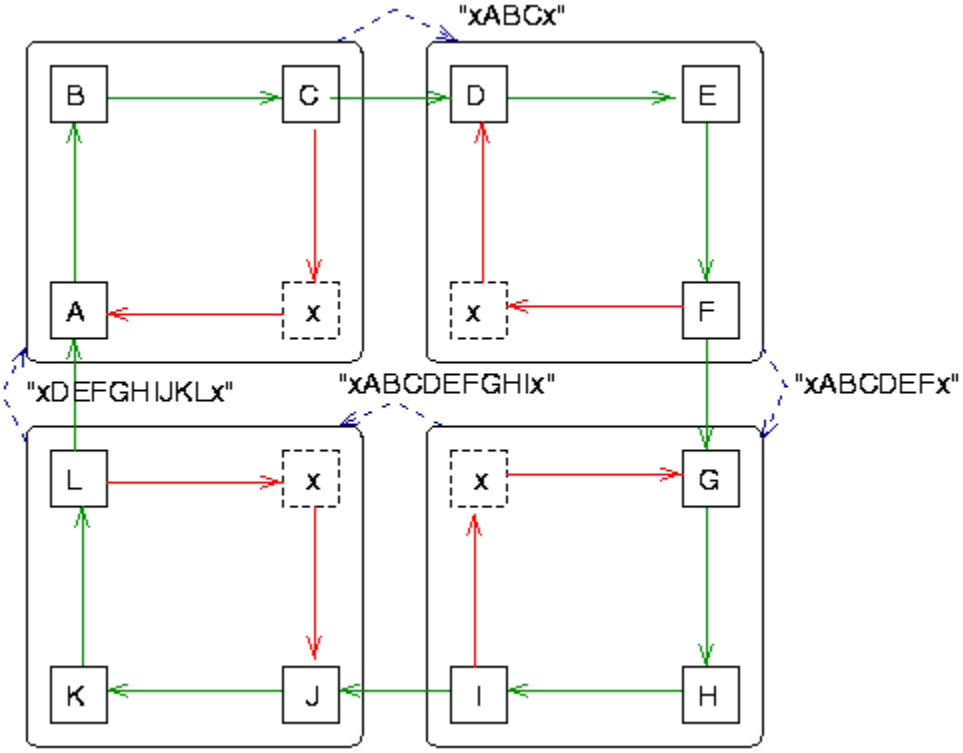


Fig 2.8 Path pushing algorithm third pass

Iteration 4:

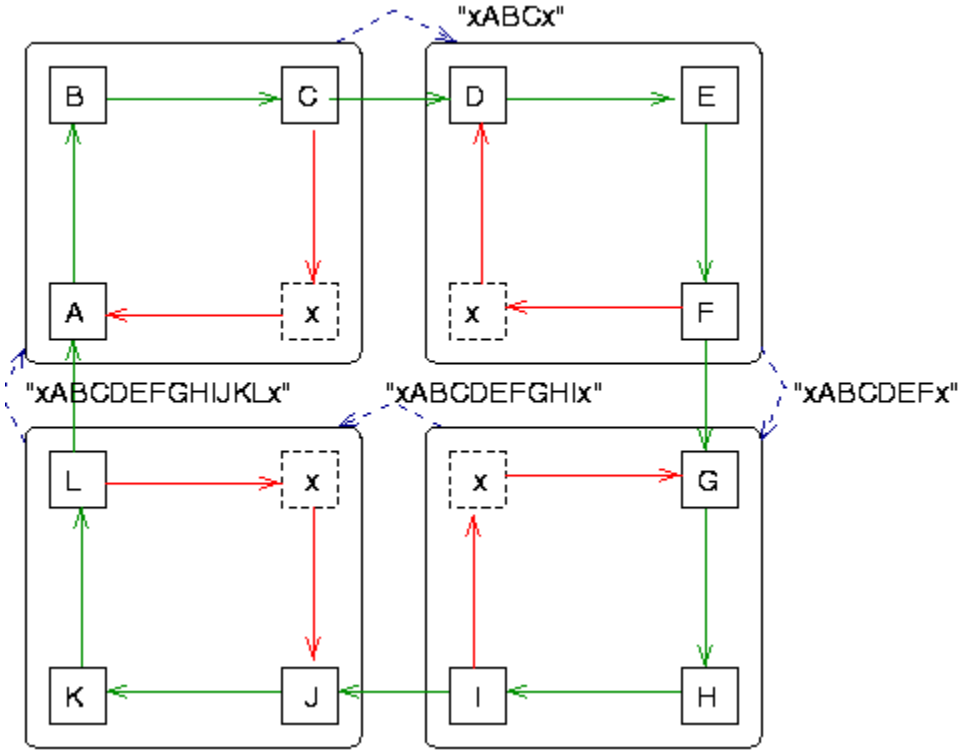


Fig 2.9 Path pushing algorithm fourth pass

Problems with Obermarck's Path-Pushing Algorithm:

- Detects false deadlocks, due to asynchronous snapshots at different sites.
- Message complexity? Message size? Detection delay?
- Exactly how are paths combined and checked?

An algorithm for detecting deadlocks in a distributed system was proposed by Chandy, Misra, and Haas in 1983. Processes request resources from the current holder of that resource. Some processes may wait for resources, which may be held either locally or remotely. Cross-machine arcs make looking for cycles, and hence detecting deadlock, difficult. This algorithm avoids the problem of constructing a Global WFG.

2. Chandy-Misra-Haas Edge-Chasing Algorithm:

The Chandy-Misra-Haas algorithm works this way: when a process has to wait for a resource, a probe message is sent to the process holding that resource. The probe message contains three components: the process ID that blocked, the process ID that is sending the request, and the destination. Initially, the first two components will be the same. When a process receives the probe: if the process itself is waiting on a resource, it updates the sending and destination fields of the message and forwards it to the resource holder. If it is waiting on multiple resources, a message is sent to each process holding the resources. This process continues as long as processes are waiting for resources. If the originator gets a message and sees its own process number in the blocked field of the message, it knows that a cycle has been taken and deadlock exists. In this case, some process (transaction) will have to die. The sender may choose to commit suicide and abort itself or an election algorithm may be used to determine an alternate victim (e.g., youngest process, oldest process,…).

Algorithm Initiation by Pi

if Pi is locally dependent on itself then declare a deadlock

else send probe (i, j, k) to home site of Pk for each j, k such that all of the following hold

- Pi is locally dependent on Pj
- Pj is waiting on Pk
- Pj and Pk are on different sites

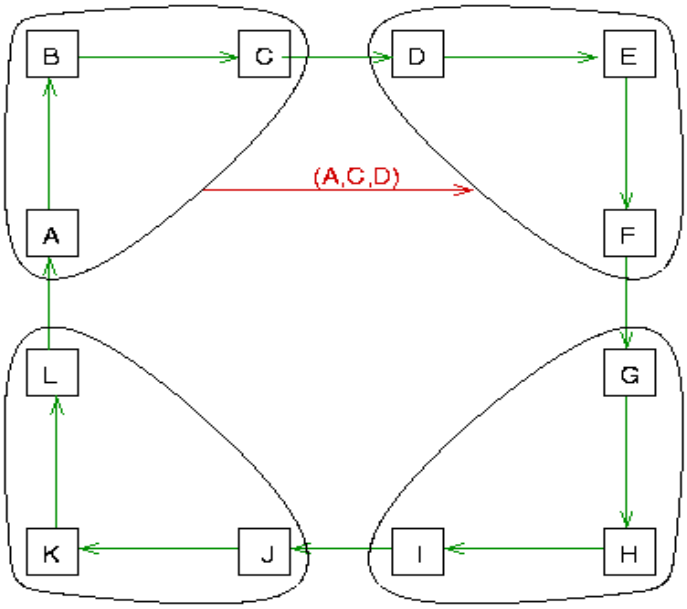


Fig 2.10 Path edge-chasing algorithm first pass

Algorithm on receipt of probe (*i*,*j*,*k*)

check the following conditions

- P_k is blocked
- $dependent_k(i) = false$
- P_k has not replied to all requests of P_j

if these are all true, do the following

- set $dependent_k(i) = true$
- if $k=i$ declare that P_i is deadlocked
- else send probe (i,m,n) to the home site of P_n for every m and n such that the following all hold
  - P_k is locally dependent on P_m
  - P_m is waiting on P_n
  - P_m and P_n are on different sites

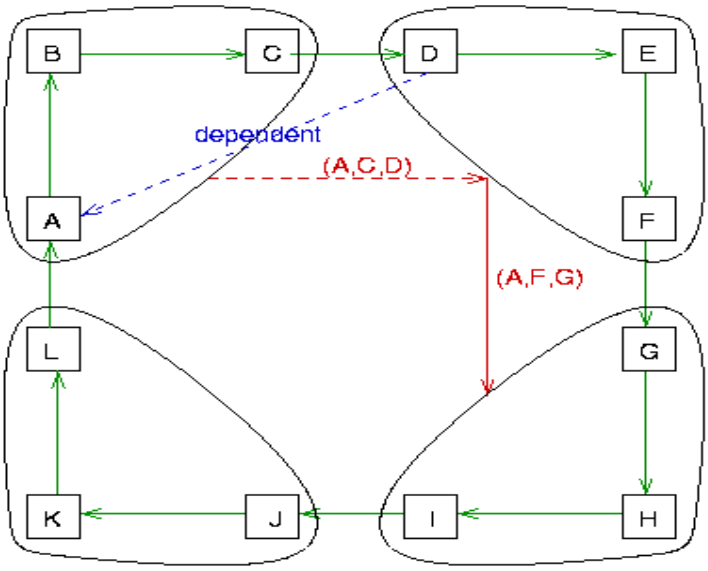


Fig 2.11 Path edge-chasing algorithm second pass

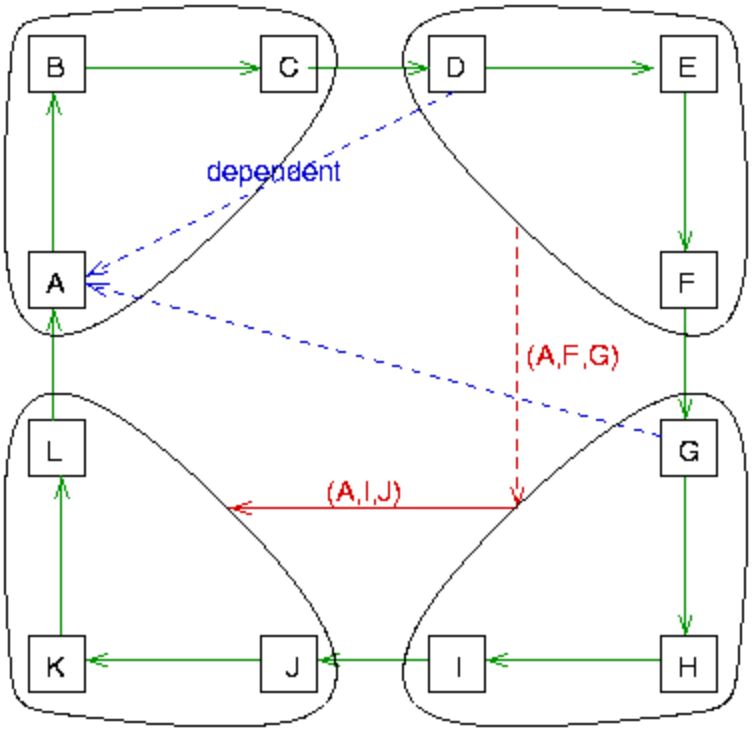


Fig 2.12 Path edge-chasing algorithm third pass

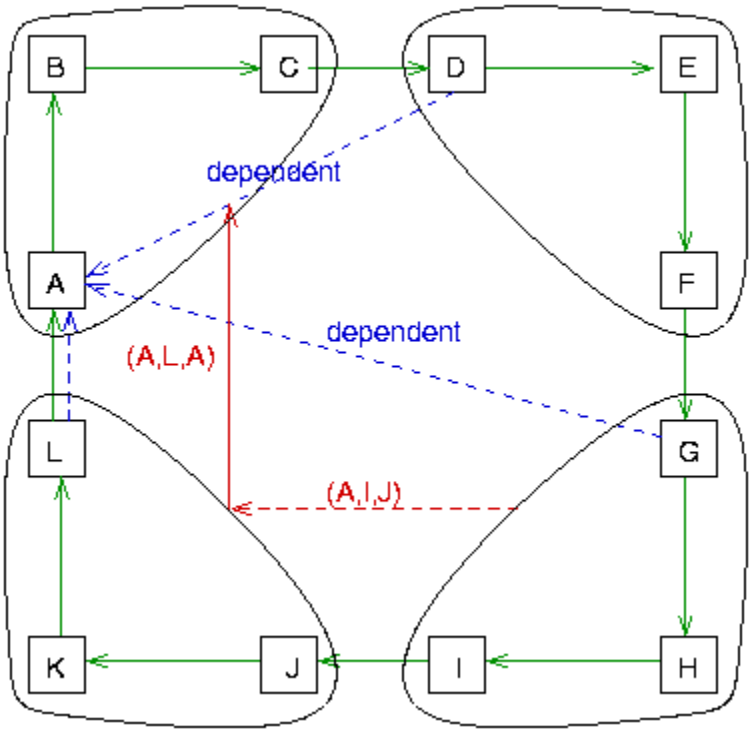


Fig 2.13 Path edge-chasing algorithm fourth pass

Agreement Protocols:

When distributed systems engage in cooperative efforts like enforcing distributed mutual exclusion algorithms, processor failure can become a critical factor. Processors may fail in various ways, and their failure modes and communication interfaces are central to the ability of healthy processors to detect and respond to such failures

It is often required that processes reach a mutual agreement. Faulty processes can send conflicting values to other processors preventing them from reaching an agreement. Processes must exchange their values and relay the values received from other processes several times to isolate the effects of faulty processes.

System Models:

There are n processors in the system and at most m of them can be faulty. The processors can directly communicate with others processors via messages (fully connected system). A receiver computation always knows the identity of a sending computation. The communication system is pipelined and reliable.

Classification of Agreement Problem

There are three well known agreement problems in distributed systems: The Byzantine agreement problem, the consensus problem and the interactive consistency problem.

In the Byzantine Agreement problem, a single value which is to be agrees on that value. One processor broadcasts a value to all other processors and all non-faulty processors agree on this value, faulty processors may agree on any (or no) value. In the consensus problem, every processor has its own initial value and all non-faulty processor must agree on a single common value. Each processor broadcasts a value to all other processors and all non-faulty processors agree on one common value from among those sent out. Faulty processors may agree on any (or no) value. In the interactive consistency problem, every processor has its own initial value and all non-faulty processor must agree on a set of common values. Each processor broadcasts a value to all other processors all non-faulty processors agree on the same set of vector of values such that vi is the initial broadcast value of non-faulty processor i . Faulty processors may agree on any (or no) value.

The Three Agreement Problems

The Byzantine agreement problem

Three generals cannot reach Byzantine agreement. An arbitrarily chosen processor, called the source processor, broadcasts its initial value to all other processors.

A solution to the Byzantine agreement problem should meet the following two objectives:

- Agreement: All non-faulty processors agree on the same value.
- Validity: If the source processor is non-faulty, then the common agreed upon value by all non-faulty processors should be initial value of the source. Two points should be noted:

1) If the source processor is faulty, then all non-faulty processors can agree on any common value.
2) It is irrelevant what value faulty processors agree on or whether they agree on a value at all.

The Consensus Problem

Every processor broadcasts its initial value to all other processors. Initial values of the processors may be different. A protocol for reaching consensus should meet the following conditions:

- Agreement: All non-faulty processors agree on the same single value.
- Validity: If the initial value of every non-faulty processor is v, then agreed upon common value by all non-faulty processor must be valid. Note that if the initial values of non-faulty processors are different, then all non-faulty processors can agree on any common value.

The Interactive Consistency Problem- Every processor broadcasts its initial value to all other processors. The initial values of the processors may be different.

A protocol for the interactive consistency problem should meet the following conditions:

- Agreement: All non-faulty processors agree on the same vector, (v1 , v2 , ..., vn).
- Validity: If the i^{th} processor is non-faulty and its initial value is vi, then the i^{th} value to be agreed on by all non-faulty processors must be v_i . Note that if the j^{th} processor is faulty, then all non-faulty processors can agree on any common value for vj .

RGPVNOTES.IN

We hope you find these notes useful.

You can get previous year question papers at https://qp.rgpvnotes.in .

If you have any queries or you want to submit your study notes please write us at rgpvnotes.in@gmail.com

LIKE & FOLLOW US ON FACEBOOK
facebook.com/rgpvnotes.in