# UNIT III

**DISTRIBUTED MUTEX & DEADLOCK**

**Distributed mutual exclusion algorithms:** Introduction – Preliminaries – Lamport„s algorithm – Ricart-Agrawala algorithm – Token-Based Algorithms – Suzuki–Kasami„s broadcast algorithm. **Deadlock detection in distributed systems:** Introduction – System model – Preliminaries – Models of deadlocks – Knapp„s classification – Algorithms for the single resource model, the AND model and the OR model.

# Mutual Exclusion 3.1.1.Introdution

* + - It ensures that concurrent process make a serialized access to shared resource or data.
    - It requires that the action performed by user on a shared resource must be atomic.
    - When a collection of process share resource then mutual exclusions needed to prevent interference and ensure consistency.
    - This problem is called critical section problem.
    - If a process, say Pi, is executing in its critical section, then no other process can be executing in their critical sections.
    - Ex., Updating DB or sending control signals to an IO devices.
    - Solution to mutual exclusion is message passing.
    - Mutual exclusion: Concurrent access of processes to a shared resource or data is executed in mutually exclusive manner.
    - Only one process is allowed to execute the critical section (CS) at any given time.
    - In a distributed system, shared variables (semaphores) or a local kernel cannot be used to implement mutual exclusion.
    - Message passing is the sole means for implementing distributed mutual exclusion.
    - One of the design issue of distributed system is Mutual exclusion.
    - Entry Section:
      * Code executed in preparation for entering the critical section
    - Critical Section:
      * Code to be protected from concurrent execution
    - Remainder Section:
      * Rest of the code

Each process cycles through these sections in the order: remainder, entry, critical, exit.

* + - Execution of critical section includes:
      * enter()
        + It is used to enter critical section-block if required.
      * resourceAccess()
        + This function is used to access shared resources.
      * exit()
        + This function is to leave the critical section so that some other process may enter.
    - Distributed mutual exclusion algorithms must deal with unpredictable message delays and incomplete knowledge of the system state.
    - Three basic approaches for distributed mutual exclusion:
      * Token based approach
      * Non-token based approach
      * Quorum based approach
    - Token-based approach:
      * A unique token is shared among the sites.
      * A site is allowed to enter its CS if it possesses the token.
      * Mutual exclusion is ensured because the token is unique
    - Non-token based approach:
      * Two or more successive rounds of messages are exchanged among the sites to determine which site will enter the CS next.
    - Quorum based approach:
      * Each site requests permission to execute the CS from a subset of sites (called a quorum).
      * Any two quorums contain a common site.
      * This common site is responsible to make sure that only one request executes the CS at any time.

# Preliminaries

* + - * System Model
        + The system consists of N sites, S1, S2, ..., SN .
        + We assume that a single process is running on each site.
        + Pi – Process @ Site Si
        + A site can be in one of the following three states:

Requesting the CS

Executing the CS,

Requesting not executing the CS (i.e., idle).

* + - * + In the „requesting the CS‟ state, the site is blocked and can not make further requests for the CS.
        + In the „idle‟ state, the site is executing outside the CS.
        + In token-based algorithms, a site can also be in a state where a site holding the token is executing outside the CS (called the idle token state).
        + At any instant, a site may have several pending requests for CS.
        + A site queues up these requests and serves them one at a time.
      * Requirements for Mutual Exclusion
        + ME1: (safety)

At most one process may execute in the critical section (CS) at a time.

* + - * + ME2: (liveness)

Requests to enter and exit the critical section eventually succeed.

This avoids starvation and deadlock

Deadlock happens when 2 or more of the processes indefinitely gets stopped when it attempts to enter or exit the critical section.

Starvation is the indefinite postponement of entry for a process that has requested it.

No starvation leads to fairness condition

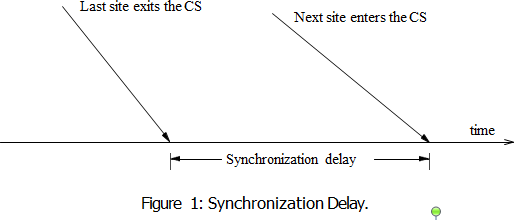
* + - * + ME3: (Ordering/Fairness)

One request to enter the critical section that happened-before another then entry to the critical section is granted in that order.

* + - * Performance Metrics
        + The performance is generally measured by the following four metrics:

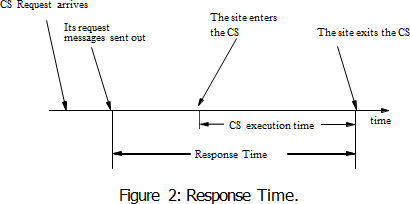
Message complexity: The number of messages required per CS execution by a site.

Synchronization delay: After a site leaves the CS, it is the time required and before the next site enters the CS (see Figure 1)



Performance Metrics

Response time: The time interval a request waits for its CS execution to be over after its request messages have been sent out (see Figure 2).



System throughput: The rate at which the system executes requests for the CS.

system throughput=1/(SD+E )

SD- synchronization delay

E - average critical section execution time.

* + - * + Low and High Load Performance:

We often study the performance of mutual exclusion algorithms under two special loading conditions, viz., “low load” and “high load”.

The load is determined by the arrival rate of CS execution requests.

Under low load conditions, there is seldom more than one request for the critical section present in the system simultaneously.

Under heavy load conditions, there is always a pending request for critical section at a site.

# Lamport’s Algorithm

* + - * Developed by Lamport
      * Requests for CS are executed in the increasing order of timestamps and time is determined by logical clocks.
      * Every site Si keeps a queue, request queuei , which contains mutual exclusion requests ordered by their timestamps.
      * This algorithm requires communication channels to deliver messages the FIFO order.
      * Algorithm:
        + Requesting the critical section:

When a site Si wants to enter the CS, it broadcasts a REQUEST(tsi , i ) message to all other sites and places the request on request queuei .

(tsi , i ) denotes the timestamp of the request.

When a site Sj receives the REQUEST(tsi , i ) message from site Si ,places site Si ‟s request on request queuej and it returns a timestamped REPLY message to Si .

* + - * + Executing the critical section: Site Si enters the CS when the following two conditions hold:

L1: Si has received a message with timestamp larger than (tsi , i ) from all other sites.

L2: Si ‟s request is at the top of request queuei .

* + - * + Releasing the critical section:

Site Si , upon exiting the CS, removes its request from the top of its request queue and broadcasts a timestamped RELEASE message to all other sites.

When a site Sj receives a RELEASE message from site Si , it removes Si ‟s request from its request queue.

When a site removes a request from its request queue, its own request may come at the top of the queue, enabling it to enter the CS.

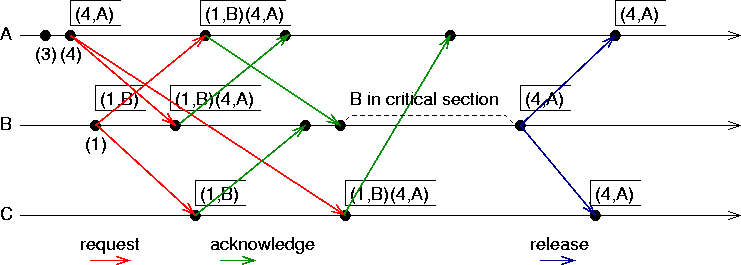
* + - * Theorem: Lamport‟s algorithm achieves mutual exclusion. Proof:
        + Proof is by contradiction.
        + Suppose two sites Si and Sj are executing the CS concurrently.
        + For this to happen conditions L1 and L2 must hold at both the sites concurrently.
        + This implies that at some instant in time, say t, both Si and Sj have their own requests at the top of their request queues and condition L1 holds at them.
        + Without loss of generality, assume that Si ‟s request has smaller timestamp than the request of Sj .
        + From condition L1 and FIFO property of the communication channels, it is clear that at instant t the request of Si must be present in request queuej when Sj was executing its CS.
        + This implies that Sj ‟s own request is at the top of its own request queue when a smaller timestamp request, Si ‟s request, is present in the request queuej – a contradiction!
      * Theorem: Lamport‟s algorithm is fair.
      * Proof:
        + The proof is by contradiction.
        + Suppose a site Si ‟s request has a smaller timestamp than the request of another site Sj and Sj is able to execute the CS before Si .
        + For Sj to execute the CS, it has to satisfy the conditions L1 and L2.
        + This implies that at some instant in time say t, Sj has its own request at the top of its queue and it has also received a message with timestamp larger than the timestamp of its request from all other sites.
        + But request queue at a site is ordered by timestamp, and according to our assumption Si has lower timestamp.
        + So Si ‟s request must be placed ahead of the Sj ‟s request in the request queuej .
        + This is a contradiction!
      * Performance
        + For each CS execution, Lamport‟s algorithm requires (N − 1) REQUEST messages, (N − 1) REPLY messages, and (N − 1) RELEASE messages.
        + Thus, Lamport‟s algorithm requires 3(N − 1) messages per CS invocation.
        + Synchronization delay in the algorithm is T .
      * Optimization
        + In Lamport‟s algorithm,REPLY messages can be omitted in certain situations.
        + For example, if site Sj receives a REQUEST message from site Si after it has sent its own REQUEST message with timestamp higher than the timestamp of site Si ‟s request, then site Sj need not send a REPLY message to site Si .
        + This is because when site Si receives site Sj ‟s request with timestamp higher than its

own, it can conclude that site Sj does not have any smaller timestamp request which is still pending.

* + - * + With this optimization, Lamport‟s algorithm requires between 3(N − 1) and 2(N −

1) messages per CS execution.

* + - * Example

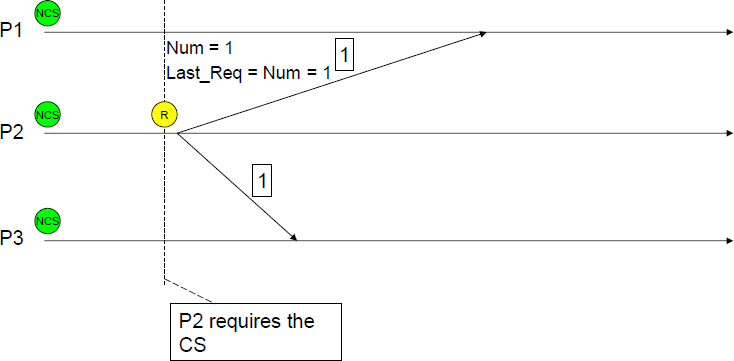


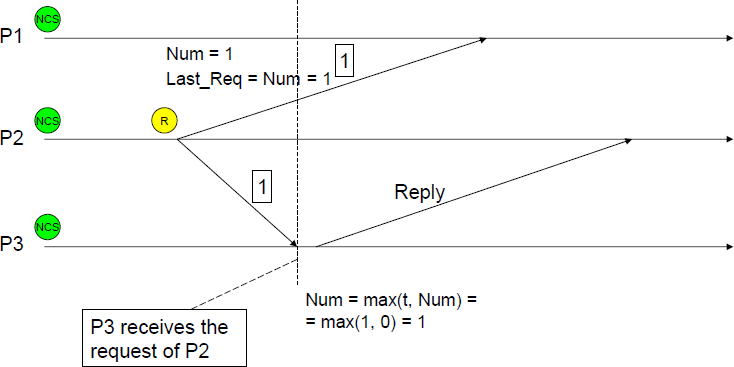
# Ricart- Agrawala Algorithm

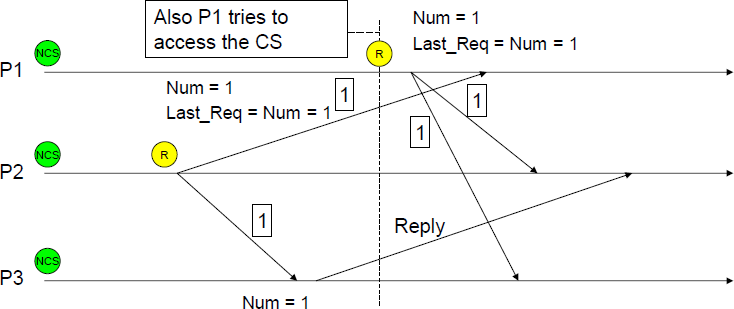
* + - * Assumes the communication channels are FIFO.
      * The algorithm uses two types of messages:
        + REQUEST
        + REPLY.
      * A process sends a REQUEST message to all other processes to request their permission to enter the critical section.
      * A process sends a REPLY message to a process to give its permission to that process.
      * Processes use Lamport-style logical clocks to assign a timestamp to critical section requests
      * Timestamps are used to decide the priority of requests.
      * Each process pi maintains the Request-Deferred array, RDi .
      * The size of which is the same as the number of processes in the system.
      * Initially, ∀i ∀j: RDi [j]=0.
      * Whenever pi defer the request sent by pj , it sets
      * RDi [j]=1 and after it has sent a REPLY message to pj , it sets RDi [j]=0. Algorithm:
      * Requesting the critical section:

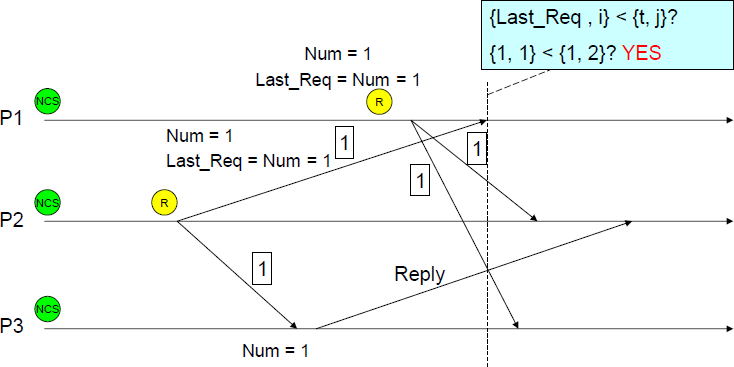
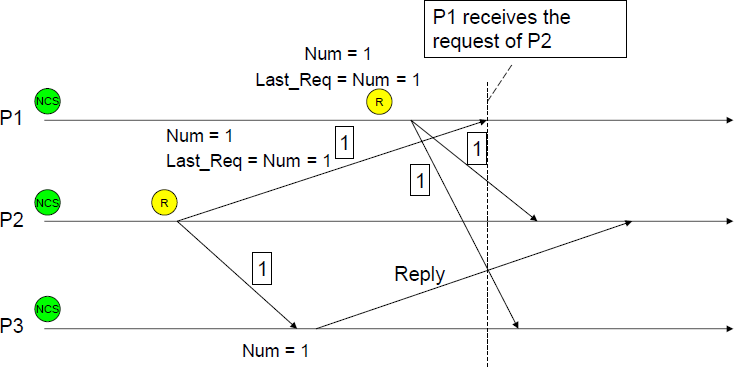
1. When a site *Si* wants to enter the CS, it broadcasts a timestamped REQUEST message to all other sites.
2. When site *Sj* receives a REQUEST message from site *Si* , it sends a REPLY message to site *Si* if site *Sj* is neither requesting nor executing the CS, or if the site *Sj* is requesting and *Si* ‟s request‟s timestamp is smaller than site *Sj* ‟s own request‟s timestamp. Otherwise, the reply is deferred and *Sj* sets *RDj* [i]=1
   * + - Executing the critical section:
3. Site Si enters the CS after it has received a REPLY message from every site it sent a REQUEST message to.
   * + - Releasing the critical section:
4. When site *Si* exits the CS, it sends all the deferred REPLY messages: ∀*j* if *RDi* [j]=1, then send a REPLY message to *Sj* and set *RDi* [j]=0.
   * + - Notes:
         * When a site receives a message, it updates its clock using the timestamp in the message.
         * When a site takes up a request for the CS for processing, it updates its local clock and assigns a timestamp to the request.
       - Correctness
         * Theorem:

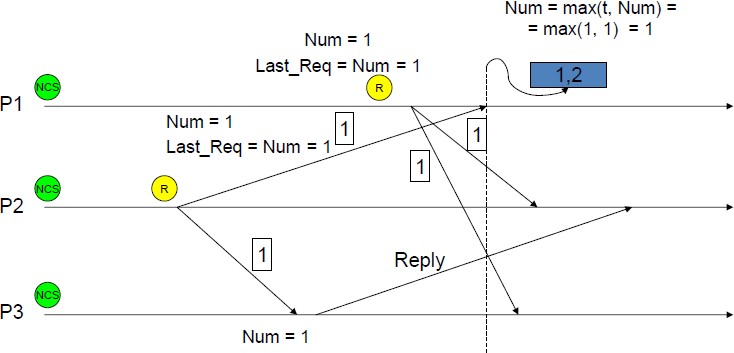
Ricart-Agrawala algorithm achieves mutual exclusion.

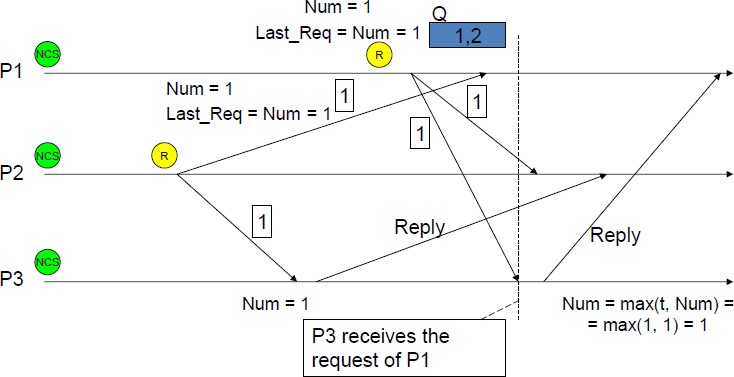
* + - * + Proof:
        + Proof is by contradiction.
        + Suppose two sites *Si* and *Sj* „ are executing the CS concurrently and *Si* ‟s request has higher priority than the request of *Sj* .
        + Clearly, *Si* received *Sj* ‟s request after it has made its own request.
        + Thus, *Sj* can concurrently execute the CS with *Si* only if *Si* returns a REPLY to *Sj* (in response to *Sj* ‟s request) before *Si* exits the CS.
        + However, this is impossible because *Sj* ‟s request has lower priority.
        + Therefore, Ricart-Agrawala algorithm achieves mutual exclusion.
      * Performance
        + For each CS execution, Ricart-Agrawala algorithm requires (N − 1) REQUEST messages and (N − 1) REPLY messages.
        + Thus, it requires 2(N − 1) messages per CS execution.
        + Synchronization delay in the algorithm is T .
* Ricart-Agrawala Algorithm Examples

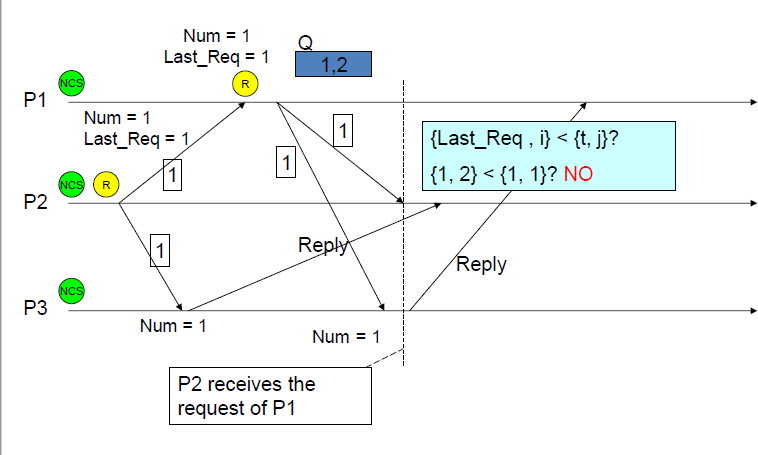


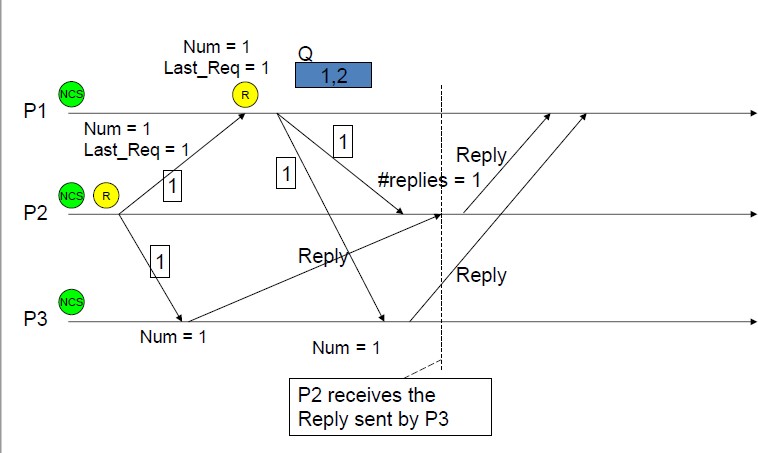
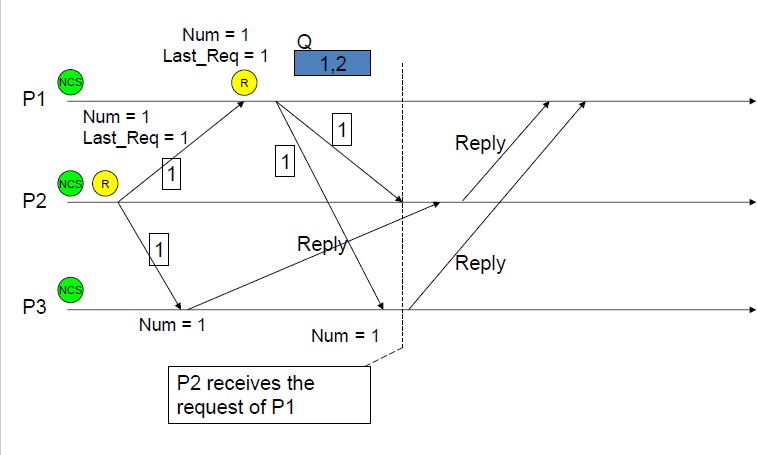


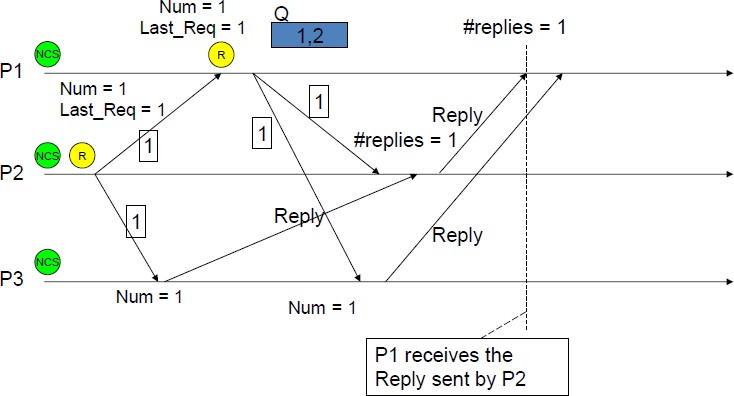


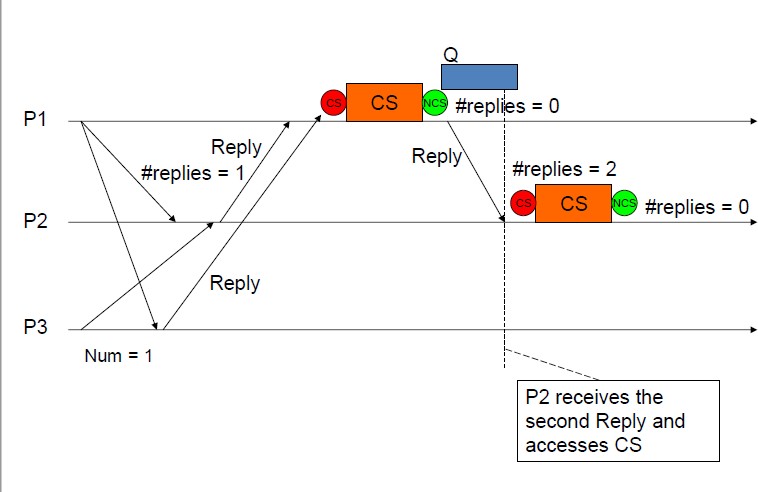
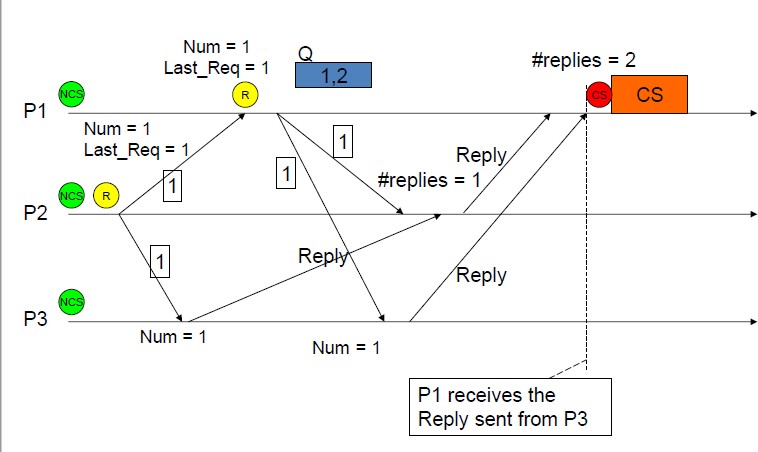












# Maekawa’s Algorithms

* + - * Quorum-Based Mutual Exclusion Algorithms
        + Quorum-based mutual exclusion algorithms are different in the following two ways:

A site does not request permission from all other sites, but only from a subset of the sites. The request set of sites are chosen such that ∀i ∀j : 1 ≤ i , j ≤ N :: Ri ∩ Rj ƒ= Φ. Consequently, every pair of sites has a site which Immediates conflicts between that pair.

A site can send out only one REPLY message at any time. A site can send a REPLY message only after it has received a RELEASE message for the previous REPLY message.

* + - * Maekawa‟s algorithm was the first quorum-based mutual exclusion algorithm.
      * The request sets for sites (i.e., quorums) in Maekawa‟s algorithm are constructed to satisfy the following conditions:
        + M1: (∀i ∀j : i ƒ= j, 1 ≤ i , j ≤ N :: Ri ∩ Rj ƒ= φ)
        + M2: (∀i : 1 ≤ i ≤ N :: Si ∈ Ri )
        + M3: (∀i : 1 ≤ i ≤ N :: |Ri | = K )
        + M4: Any site Sj is contained in K number of Ri s, 1 ≤ i , j ≤ N .
      * Maekawa used the theory of projective planes and showed that N = K (K − 1) + 1.
      * This relation gives |Ri | = N.
      * Conditions M1 and M2 are necessary for correctness
      * Conditions M3 and M4 provide other desirable features to the algorithm.
      * Condition M3 states that the size of the requests sets of all sites must be equal implying that all sites should have to do equal amount of work to invoke mutual exclusion.
      * Condition M4 enforces that exactly the same number of sites should request permission from any site implying that all sites have “equal responsibility” in granting permission to other sites.
      * A site Si executes the following steps to execute the CS. Algorithm:
      * Requesting the critical section
        + (a) A site Si requests access to the CS by sending REQUEST(i ) messages to all sites in its request set Ri .
        + (b) When a site Sj receives the REQUEST(i ) message, it sends a REPLY(j) message to Si provided it hasn‟t sent a REPLY message to a site since its receipt of the last RELEASE message. Otherwise, it queues up the REQUEST(i) for later consideration.
      * Executing the critical section
        + (c) Site Si executes the CS only after it has received a REPLY message from every site in Ri .
      * Releasing the critical section
        + (d) After the execution of the CS is over, site Si sends a RELEASE(i ) message to every site in Ri .
        + (e) When a site Sj receives a RELEASE(i ) message from site Si , it sends a REPLY message to the next site waiting in the queue and deletes that entry from the queue. If the queue is empty, then the site updates its state to reflect that it has not sent out any REPLY message since the receipt of the last RELEASE message.
      * Correctness
        + Theorem: Maekawa‟s algorithm achieves mutual exclusion.
        + Proof:

Proof is by contradiction. Suppose two sites Si and Sj are concurrently executing the CS.

This means site Si received a REPLY message from all sites in Ri and concurrently site Sj was able to receive a REPLY message from all sites in Rj

.

If Ri ∩ Rj = {Sk }, then site Sk must have sent REPLY messages to both Si and Sj concurrently, which is a contradiction. Q

* + - * Performance
        + Size of request set is √N
        + An execution of the CS requires √ N REQUEST, √ N REPLY, and √ N RELEASE messages, resulting in 3 √ N messages per CS execution.
        + Synchronization delay in this algorithm is 2T.
      * Problem of Meakawas Algorithms
      * Maekawa‟s algorithm can deadlock because a site is exclusively locked by other sites and requests are not prioritized by their timestamps.
      * Assume three sites Si , Sj , and Sk simultaneously invoke mutual exclusion. Suppose Ri ∩ Rj = {Sij }, Rj ∩ Rk = {Sjk }, and Rk ∩ Ri = {Ski }.
      * Consider the following scenario:
        + Sij has been locked by Si (forcing Sj to wait at Sij ).
        + Sjk has been locked by Sj (forcing Sk to wait at Sjk ).
        + Ski has been locked by Sk (forcing Si to wait at Ski ).
      * This state represents a deadlock involving sites Si , Sj , and Sk .
      * Handling Deadlocks
        + Maekawa‟s algorithm handles deadlocks by requiring a site to yield a lock if the timestamp of its request is larger than the timestamp of some other request waiting for the same lock.
        + A site suspects a deadlock (and initiates message exchanges to resolve it) whenever a higher priority request arrives and waits at a site because the site has sent a REPLY message to a lower priority request.
        + Deadlock handling requires three types of messages:

FAILED: A FAILED message from site Si to site Sj indicates that Si can not grant Sj ‟s request because it has currently granted permission to a site with a higher priority request.

INQUIRE: An INQUIRE message from Si to Sj indicates that Si would like to find out from Sj if it has succeeded in locking all the sites in its request set.

YIELD: A YIELD message from site Si to Sj indicates that Si is returning the permission to Sj (to yield to a higher priority request at Sj ).

* + - * Maekawa‟s algorithm handles deadlocks as follows:
        + When a REQUEST(ts, i ) from site Si blocks at site Sj because Sj has currently granted permission to site Sk , then Sj sends a FAILED(j) message to Si if Si ‟s request has lower priority. Otherwise, Sj sends an INQUIRE(j) message to site Sk .
        + In response to an INQUIRE(j) message from site Sj , site Sk sends a YIELD(k ) message to Sj provided Sk has received a FAILED message from a site in its request set or if it sent a YIELD to any of these sites, but has not received a new GRANT from it.
        + In response to a YIELD(k ) message from site Sk , site Sj assumes as if it has been released by Sk , places the request of Sk at appropriate location in the request queue, and sends a GRANT(j) to the top request‟s site in the queue. Maekawa‟s algorithm requires extra messages to handle deadlocks
        + Maximum number of messages required per CS execution in this case is 5√N

# Suzuki- Kasami’s Broadcast Algorithm

* + - * Token-Based Algorithms
        + In token-based algorithms, a unique token is shared among the sites.
        + A site is allowed to enter its CS if it possesses the token.
        + Token-based algorithms use sequence numbers instead of timestamps. (Used to distinguish between old and current requests.)
        + Ex: Suzuk- Kasami‟s Broadcast Algorithm
      * If a site wants to enter the CS and it does not have the token, it broadcasts a REQUEST message for the token to all other sites.
      * A site which possesses the token sends it to the requesting site upon the receipt of its REQUEST message.
      * If a site receives a REQUEST message when it is executing the CS, it sends the token only after it has completed the execution of the CS.
      * Algorithm:
        + Requesting the critical section

(a) If requesting site Si does not have the token, then it increments its sequence number, RNi [i], and sends a REQUEST(i, sn) message to all other sites. („sn‟ is the updated value of RNi [i].)

(b) When a site Sj receives this message, it sets RNj [i] to max(RNj [i], sn). If Sj has the idle token, then it sends the token to Si if RNj [i]=LN[i]+1.

* + - * + Executing the critical section

(c) Site Si executes the CS after it has received the token.

* + - * + Releasing the critical section

(d)It sets LN[i] element of the token array equal to RNi [i].

(e)For every site Sj whose id is not in the token queue, it appends its id to the token queue if RNi [j]=LN[j]+1.

(f)If the token queue is nonempty after the above update, Si deletes the top site id from the token queue and sends the token to the site indicated by the id.

* + - * Design Issues:
        + This algorithm must addresses the following design issues:

How to distinguish an outdated REQUEST message from a current REQUEST message?

Due to variable message delays, a site may receive a token request message after the corresponding request has been satisfied.

If a site can not determined if the request corresponding to a token request has been satisfied, it may dispatch the token to a site that does not need it.

This will not violate the correctness, however, this may seriously degrade the performance.

How to determine which site has an outstanding request for the CS?

After a site has finished the execution of the CS, it must determine what sites have an outstanding request for the CS so that the token can be dispatched to one of them.

* + - * Solution for first issue:
        + A REQUEST message of site Sj has the form REQUEST(j, n) where n (n=1, 2, ...) is a sequence number
        + It indicates that site Sj is requesting its nth CS execution.
        + A site Si keeps an array of integers RNi [1..N] where RNi [j] denotes the largest sequence number received in a REQUEST message so far from site Sj .
        + When site Si receives a REQUEST(j, n) message, it sets RNi [j]:= max(RNi [j], n).
        + When a site Si receives a REQUEST(j, n) message, the request is outdated if RNi [j]>n.
      * Solution for second issue:
        + The token consists of a queue of requesting sites, Q, and an array of integers LN[1..N], where LN[j] is the sequence number of the request which site Sj executed most recently.
        + After executing its CS, a site Si updates LN[i]:=RNi [i] to indicate that its request corresponding to sequence number RNi [i] has been executed.
        + At site Si if RNi [j]=LN[j]+1, then site Sj is currently requesting token.
      * Correctness
        + Mutual exclusion is guaranteed because there is only one token in the system and a site holds the token during the CS execution.

Theorem: A requesting site enters the CS in finite time.

Proof:

Token request messages of a site Si reach other sites in finite time.

Since one of these sites will have token in finite time, site Si ‟s request will be placed in the token queue in finite time.

Since there can be at most N − 1 requests in front of this request in the token queue, site Si will get the token and execute the CS in finite time.

* + - * Performance
        + No message is needed and the synchronization delay is zero if a site holds the idle token at the time of its request.
        + If a site does not hold the token when it makes a request, the algorithm requires N messages to obtain the token. Synchronization delay in this algorithm is 0 or T .

# Deadlock Detection in Distributed Systems

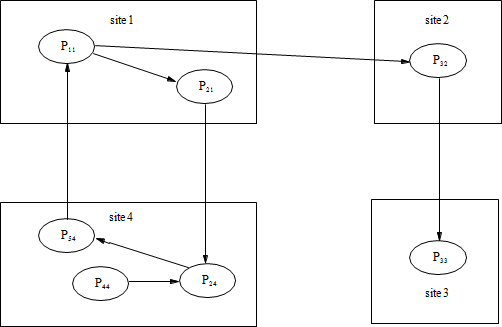
* + 1. **Introduction**
       - Deadlocks is a fundamental problem in distributed systems.
       - A process may request resources in any order, which may not be known a priori and a process can request resource while holding others.
       - If the sequence of the allocations of resources to the processes 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

* + - * A distributed program is composed of a set of n asynchronous processes p1, p2, . . . , p*i* , . . .

, p*n* 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.
      * 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:
        + The systems have only reusable resources.
        + Processes are allowed to make only exclusive access to resources.
        + There is only one copy of each resource
      * Each process utilizes a resources as follows:-
        + Request
        + Use
        + Release
      * A process can be in two states: -
        + Running
        + 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.
      * Wait-For-Graph(WFG):
        + The state of the system can be modeled by directed graph, called a wait for graph (WFG).
        + In a WFG , nodes are processes and there is a directed edge from node P1 to node P2 if P1 is blocked and is waiting for P2 to release some resource.
        + A system is deadlocked if and only if there exists a directed cycle or knot in the WFG.
        + The following figure shows a WFG.
        + Here, process P11 of site 1 has an edge to process P21 of site 1 and P32 of site 2 is waiting for a resource which is currently held by process P21.
        + At the same time process P32 is waiting on process P33 to release a resource.
        + If P21 is waiting on process P11, then processes P11, P32 and P21 form a cycle and all the four processes are involved in a deadlock depending upon the request model.





# Preliminaries

* + - * Deadlock Handling Strategies
        + There are three strategies for handling deadlocks:

Deadlock prevention

Deadlock avoidance

Deadlock detection

* + - * Handling of deadlock becomes highly complicated in distributed systems.
      * Because no site has accurate knowledge of the current state of the system.
      * And also, every inter-site communication involves a finite and unpredictable delay.
        + Deadlock prevention is commonly achieved either by having a process acquire all the needed resources simultaneously before it begins executing or by preempting a process which holds the needed resource.
        + This approach is highly inefficient and impractical in distributed systems.
        + Deadlock avoidance approach is defined as a resource is granted to a process if the resulting global system state is safe.

However, due to several problems, deadlock avoidance is impractical in distributed systems.

Deadlock detection:

Principle: detection of a cycle in WFG proceeds concurrently with normal operation.

Detection Algorithm must detect all existing deadlock in finite time.

Detection algorithm should not report non-existent (phantom) deadlock.

**Resolution**(recovery):

All existing wait-for dependencies in WFG must be removed and give their resources to other blocked processes.

* + - * Issues in Deadlock Detection
        + Two basic issues:

First, detection of existing deadlocks

Second, resolution of detected deadlocks.

* + - * + Detection of deadlocks involves addressing two issues:

Maintenance of the WFG

Searching of the WFG for the presence of cycles (or knots).

* + - * Correctness Criteria:
        + A deadlock detection algorithm must satisfy the following two conditions:

Progress (No undetected deadlocks):

The algorithm must detect all existing deadlocks in finite time.

In other words, after all wait-for dependencies for a deadlock have formed, the algorithm should not wait for any more events to occur to detect the deadlock.

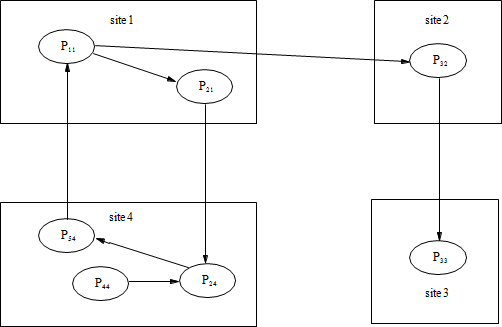
Safety (No false deadlocks):

The algorithm should not report deadlocks which do not exist (called phantom or false deadlocks).

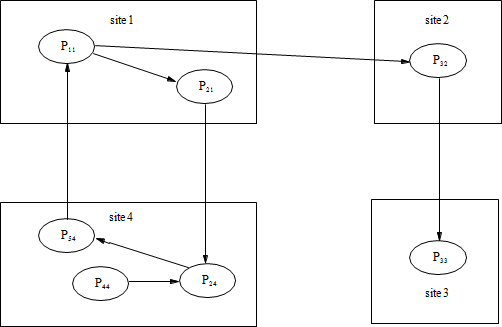
* + - * Resolution of a Detected Deadlock:
        + Deadlock resolution involves breaking existing wait-for dependencies between the processes to resolve the deadlock.
        + It involves rolling back one or more deadlocked processes and assigning their resources to blocked processes so that they can resume execution.

# Models of Deadlock

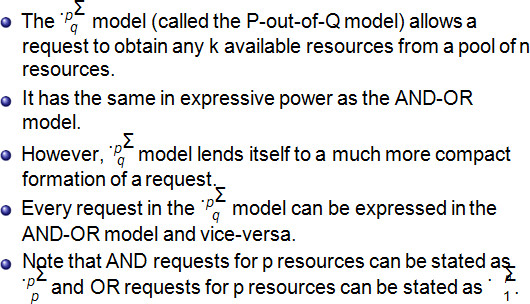
* + - * Distributed systems allow several kinds of resource requests.
      * The Single Resource Model:
        + In the single resource model, a process can have at most one outstanding request for only one unit of a resource.
        + Since the maximum out-degree of a node in a WFG for the single resource model can be 1, the presence of a cycle in the WFG shall indicate that there is a deadlock.
      * The AND Model:
        + In the AND model, a process can request for more than one resource simultaneously and the request is satisfied only after all the requested resources are granted to the process.
        + The out degree of a node in the WFG for AND model can be more than 1.
        + The presence of a cycle in the WFG indicates a deadlock in the AND model.
        + Since in the single-resource model, a process can have at most one outstanding request, the AND model is more general than the single-resource model.
        + Consider the example WFG described in the Figure.
        + P11 has two outstanding resource requests. In case of the AND model, P11shall become active from idle state only after both the resources are granted.
        + There is a cycle P11->P21->P24->P54->P11 which corresponds to a deadlock situation.
        + That is, a process may not be a part of a cycle, it can still be deadlocked. Consider process P44 in Figure.
        + It is not a part of any cycle but is still deadlocked as it is dependent on P24which is deadlocked.



* + - * The OR Model
        + In the OR model, a process can make a request for numerous resources simultaneously and the request is satisfied if any one of the requested resources is granted.
        + Presence of a cycle in the WFG of an OR model does not imply a deadlock in the OR model.
        + Consider example in Figure ,If all nodes are OR nodes, then process P11 is not deadlocked because once process P33 releases its resources, P32 shall become active as one of its requests is satisfied.
        + After P32 finishes execution and releases its resources, process P11 can continue with its processing.
        + In the OR model, the presence of a knot indicates a deadlock.



* + - * The AND-OR Model:
        + A generalization of the previous two models (OR model and AND model) is the AND-OR model.
        + In the AND-OR model, a request may specify any combination of *and* and *or* in the resource request.
        + For example, in the AND-OR model, a request for multiple resources can be of the form x *and* (y *or* z).
        + To detect the presence of deadlocks in such a model, there is no familiar construct of graph theory using WFG.
        + Since a deadlock is a stable property, a deadlock in the AND-OR model can be detected by repeated application of the test for OR-model deadlock.
      * The .*pq*Σ Model:



* + - * Unrestricted Model
        + In the unrestricted model, no assumptions are made regarding the underlying structure of resource requests.
        + Only one assumption that the deadlock is stable and hence it is the most general model.
        + This model helps separate concerns:

Concerns about properties of the problem (stability and deadlock) are separated from underlying distributed systems computations (e.g., message passing versus synchronous communication).

# Knapps Classification

* + - * Distributed deadlock detection algorithms can be divided into four classes:
        + Path-Pushing
        + Edge-Chasing
        + Diffusion Computation
        + Global State Detection

# Path-Pushing Algorithms

* + - * In path-pushing deadlock detection algorithms, information about the wait-for dependencies is propagated in the form of paths.
      * Obermarck‟s algorithm for path propagation is described here i.e. an AND model.
      * It is based on a database model using transaction processing.
      * Sites which detect a cycle in their partial WFG views convey the paths discovered to members of the transaction.
      * The highest priority transaction detects the deadlock “Ex=>T1=>T2=>Ex”
      * Algorithm can detect phantoms due to its asynchronous snapshot methods.

# Obermarck’s Algorithm

* + - * 1. The site waits for deadlock-related information from other sites.
      * 2. The site combines the received information with its local TWF graph to build an updated TWF graph.
      * 3. It then detects all cycles and break only those cycles which do not contain the node „Ex‟.
      * 4. For all cycles „Ex T1 T2 Ex‟ which contain the node ‟Ex‟, the site transmits them in string from „Ex ,T1,T2, Ex‟ to all other sites where an agent of T2 is waiting for a resource being held by another transaction.
      * 5. The algorithm reduces message traffics by lexically ordering transactions and sending the string „Ex ,T1,T2 ,T3,Ex‟ to other sites only if T1 is higher than T3 in a lexical ordering.
      * 6. For a deadlock, the highest priority transaction detects the deadlock.

# Features:

* + - * + The non local portion of the global Transaction Wait-For (TWF) graph at a site is abstracted by a distinguished node which helps in determining potential multisite deadlocks without requiring a huge global TEF graph to be stored at each site.
        + Transactions are totally ordered. It also ensures that exactly one transaction in each cycle detects the deadlock.

# Edge-Chasing Algorithms

* + - * In an edge-chasing algorithm, the presence of a cycle in a distributed graph structure is be verified by propagating special messages called probes, along the edges of the graph.
      * These probe messages are different than the request and reply messages.
      * The formation of cycle can be deleted by a site if it receives the matching probe sent by it previously.
      * Whenever a process that is executing receives a probe message, it discards this message and continues.
      * Only blocked processes propagate probe messages along their outgoing edges.
      * Main advantage of edge-chasing algorithms is that probes are fixed size messages which is normally very short.

Algorithm

* + - * if *Pi* is locally dependent on itself then declare a deadlock else for all *Pj* and *Pk* such that
        + *Pi* is locally dependent upon *Pj* , and
        + *Pj* is waiting on *Pk* , and
        + *Pj* and *Pk* are on different sites,send a probe (i, j, k) to the home site of *Pk*
      * On the receipt of a probe (i, j, k), the site takes the following actions:
        + If
        + *Pk* is blocked, and
        + *dependentk* (i) is false, and
        + *Pk* has not replied to all requests *Pj*
      * Then
        + Begin
        + *dependentk* (i) = true; if k=i

then declare that *Pi* is deadlocked else for all *Pm* and *Pn* such that

(a‟) *Pk* is locally dependent upon *Pm*, and

(b‟) *Pm* is waiting on *Pn*, and

*Pm* and *Pn* are on different sites, send a probe (i, m, n) to the home site of *Pn*

End.

# Diffusing Computations Based Algorithms

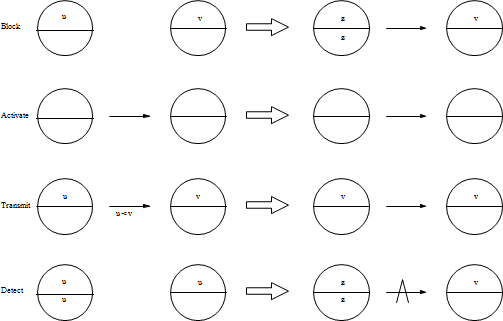
* + - * In *diffusion computation* based distributed deadlock detection algorithms, deadlock detection computation is diffused through the WFG of the system.
      * These algorithms make use of echo algorithms to detect deadlocks.
      * This computation is superimposed on the underlying distributed computation. If this computation terminates, the initiator declares a deadlock.
      * To detect a deadlock, a process sends out query messages along all the outgoing edges in the WFG.
      * These queries are successively propagated (i.e., diffused) through the edges of the WFG.

# Global State Detection

* + - * Based on two facts of distributed systems:
        + A consistent snapshot of a distributed system can be obtained without freezing the underlying computation.
        + A consistent snapshot may not represent the system state at any moment in time, but if a stable property holds in the system before the snapshot collection is initiated, this property will sill hold in the snapshot

# Algorithms for the single resource model

* + - * Mitchell and Merritt‟s Algorithm for the Single-Resource Model
        + Belongs to the class of edge-chasing algorithms where probes are sent in opposite direction of the edges of WFG.
        + When a probe initiated by a process comes back to it, the process declares deadlock.
        + Only one process in a cycle detects the deadlock. This simplifies the deadlock resolution – this process can abort itself to resolve the deadlock.
      * Each node of the WFG has two local variables, called labels:
        + Private label: it is unique to the node at all times, though it is not constant
        + Public label: it cab read by other processes and it may not be unique.
      * Each process is represented as u/v where u and u are the public and private labels, respectively.
      * Initially, private and public labels are equal for each process.
      * A global WFG is maintained and it defines the entire state of the system**.**
      * The algorithm is defined by the four state transitions shown in Figure, where z = inc(u, v), and inc(u, v) yields a unique label greater than both u and v labels that are not shown do not change.
      * Block creates an edge in the WFG.
      * Two messages are needed, one resource request and one message back to the blocked process to inform it of the public label of the process it is waiting for.
      * Activate denotes that a process has acquired the resource from the process it was waiting for.
      * Transmit propagates larger labels in the opposite direction of the edges by sending a probe message.





* + - * Whenever a process receives a probe which is less then its public label, then it simply ignores that probe.
      * Detect means that the probe with the private label of some process has returned to it, indicating a deadlock.
      * The above algorithm can be easily extended to include priorities where whenever a deadlock occurs, the lowest priority process gets aborted.
      * Message Complexity:
        + If we assume that a deadlock persists long enough to be detected, the worst-case complexity of the algorithm is s(s - 1)/2 Transmit steps, where s is the number of processes in the cycle.

# The AND model and OR model.

* + - * **Chandy-Misra-Haas Algorithm for the AND Model:**
        + Chandy-Misra-Haas‟s distributed deadlock detection algorithm for AND model is based on edge-chasing.
        + The algorithm uses a special message called probe, which is a triplet (i, j, k), denoting that it belongs to a deadlock detection initiated for process Pi and it is being sent by the home site of process Pj to the home site of process Pk .
        + A probe message travels along the edges of the global WFG graph, and a deadlock is detected when a probe message returns to the process that initiated it.
        + A process Pj is said to be dependent on another process Pk if there exists a sequence of processes Pj , Pi 1, Pi 2, ..., Pim, Pk such that each process except Pk in the sequence is blocked and each process, except the Pj , holds a resource for which the previous process in the sequence is waiting.
        + Process Pj is said to be locally dependent upon process Pk if Pj is dependent upon Pk and both the processes are on the same site.
        + Data Structures

Each process Pi maintains a boolean array, dependenti , where dependenti (j) is true only if Pi knows that Pj is dependent on it.

Initially, dependenti (j) is false for all i and j.

* + - * Algorithm:
        + The following algorithm determines if a blocked process is deadlocked:

if Pi is locally dependent on itself then declare a deadlock else for all Pj and Pk such that

Pi is locally dependent upon Pj , and

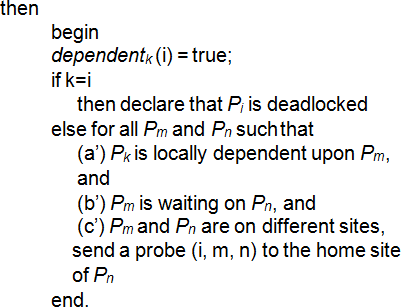
Pj is waiting on Pk , and Pj and Pk are on different sites,send a probe (i, j, k) to the home site of Pk

* + - * + On the receipt of a probe (i, j, k), the site takes the following actions: if

Pk is blocked, and

dependentk (i) is false, and

Pk has not replied to all requests Pj ,



* + - * + A probe message is continuously circulated along the edges of the global WFG graph and a deadlock is detected when a probe message returns to its initiating process.
        + Performance Analysis

One probe message (per deadlock detection initiation) is sent on every edge of the WFG which that two sites.

Thus, the algorithm exchanges at most m(n − 1)/2 messages to detect a deadlock that involves m processes and that spans over n sites.

The size of messages is fixed and is very small (only 3 integer words).

Delay in detecting a deadlock is O(n).

# Chandy-Misra-Haas Algorithm for the OR Model:

* + - * + Chandy-Misra-Haas distributed deadlock detection algorithm for OR model is based on the approach of diffusion- omputation.

A blocked process determines if it is deadlocked by initiating a diffusion computation.

Two types of messages are used in a diffusion computation:

query(i, j, k) and reply(i, j, k), denoting that they belong to a diffusion computation initiated by a process Pi and are being sent from process Pj to process Pk .

* + - * + A blocked process initiates deadlock detection by sending query messages to all processes in its dependent set.
        + If an active process receives a query or reply message, it discards it.
        + When a blocked process Pk receives a query(i, j, k) message, it takes the following actions:

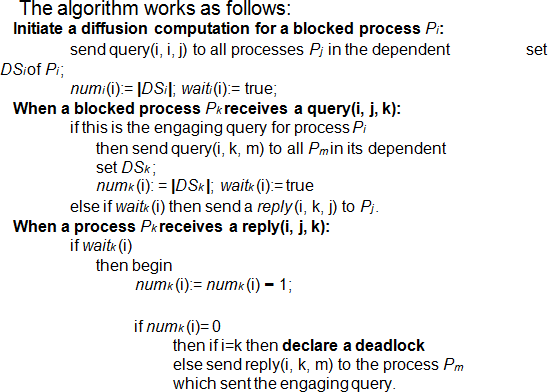
If this is the first query message received by Pk for the deadlock detection initiated by Pi (called the engaging query), then it propagates the query to all the processes in its dependent set and sets a local variable numk (i) to the number of query messages sent.

If this is not the engaging query, then Pk returns a reply message to it immediately provided Pk has been continuously blocked since it received the corresponding engaging query. Otherwise, it discards the query.

* + - * + Process Pk maintains a boolean variable waitk (i) that denotes the fact that it has been continuously blocked since it received the last engaging query from process Pi

.

* + - * + When a blocked process Pk receives a reply(i, j, k) message, it decrements numk (i) only if waitk (i) holds.
        + A process sends a reply message in response to an engaging query only after it has received a reply to every query message it had sent out for this engaging query.
        + The initiator process detects a deadlock when it receives reply messages to all the query messages it had sent out.
      * Algorithm



* + - * + In practice, several diffusion computations may be initiated for a process (A diffusion computation is initiated every time the process gets blocked), but, at any time only one diffusion computation is current for any process.
        + However, messages for outdated diffusion computations may still be in transit.
        + The current diffusion computation can be distinguished from outdated ones by using sequence numbers.

Performance Analysis

For every deadlock detection, the algorithm exchanges e query messages and e reply messages, where e=n(n-1) is the number of edges.