Synchronization Issues

- Sharing of resources among concurrently executing processes requires synchronization mechanisms to ensure that this happens correctly
- Distributed systems need to deal with the following synchronization-related issues:
  - Clock synchronization
  - Event ordering
  - Mutual exclusion
  - Deadlock
  - Elections

Clock Synchronization

Why Clock Synchronization?

- Every computer has a built-in clock
- Different clocks have different drift rates
- Hence, with the passage of time, a computer clock drifts from the real-time clock that was used for its initial setting
- Hence, a computer clock must be periodically resynchronized with the real-time clock to keep it non-faulty

Clock Synchronization Needs in Distributed Systems

- In distributed systems, the following types of clock synchronization are needed:
  - Synchronization of computer clocks with real-time (or external) clocks
  - Mutual (or internal) synchronization of the clocks of different nodes
- Mutual synchronization of the clocks of different nodes is needed for:
  - Correct results of distributed applications
  - Measuring the duration of distributed activities, which start on one node and terminate on another node

Clock Synchronization Issues

- A set of clocks are said to be synchronized if the clock skew of any two clocks in this set is less than some specified constant δ.
- An important issue in clock synchronization is that time must never run backward.
- Externally synchronized clocks are also internally synchronized, but the converse is not true

Clock Synchronization Algorithms

- Centralized algorithms
  - One node (called the time server node) has a real-time receiver. It's time is used as the reference time.
  - The clocks of all other nodes are synchronized with the clock time of the time server node.
- Distributed algorithms
  - Each node has a real-time receiver for external synchronization
  - Nodes communicate with each other for internal synchronization

Centralized Clock Synchronization Algorithms

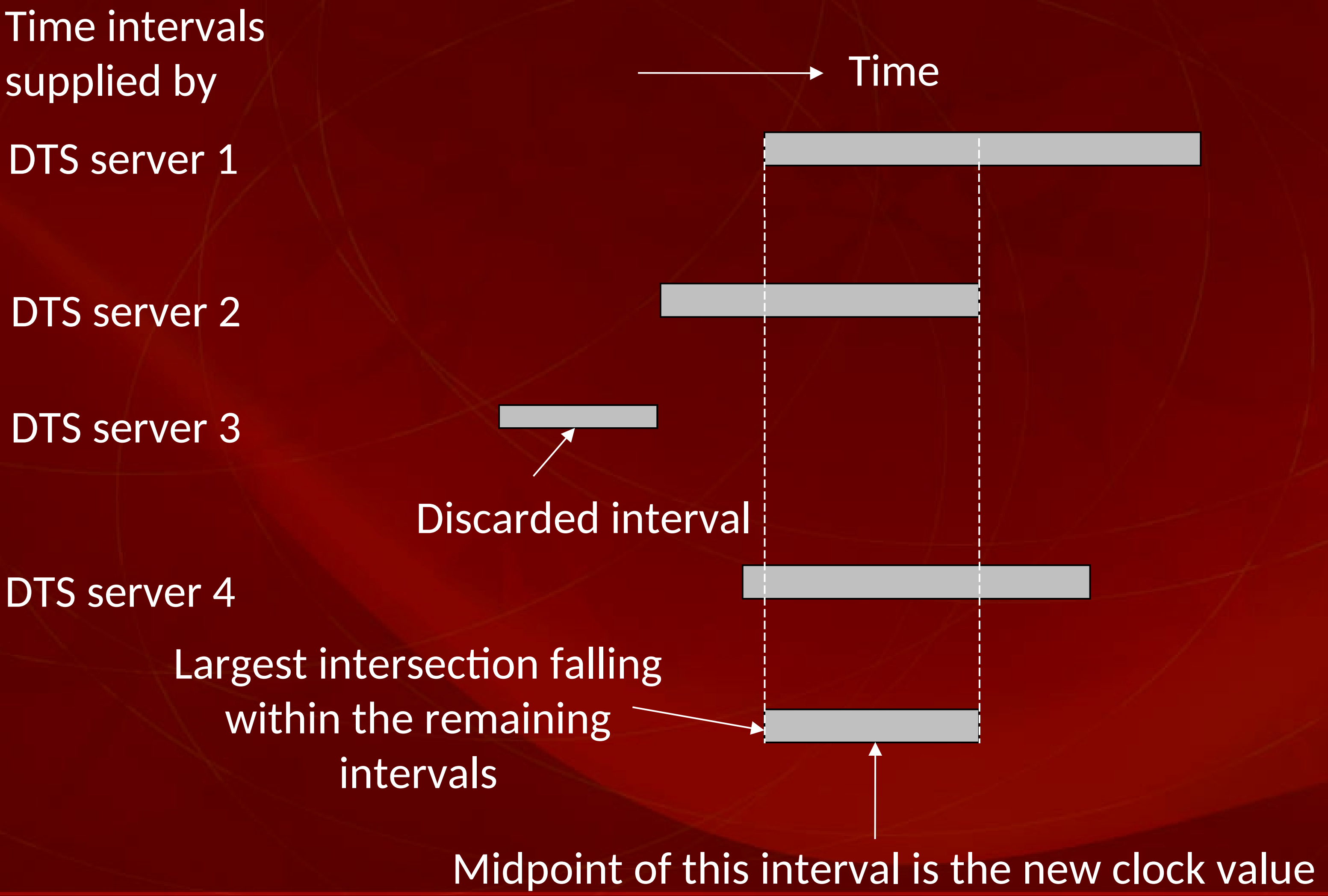
- *Passive Time Server Algorithm*
  - Each node periodically sends a message (time = ?).
  - The time server responds with a message (time = T)
  - On receiving the message, the node readjusts its clock time to *T + (T1 – T0) / 2*
- *Active Time Server Algorithm*
  - The time server periodically broadcasts its clock time (time = T)
  - On receiving the message, a node readjusts its clock time to $T + T_a$

Distributed Time Service (DTS)

- It is used to synchronize clocks of a distributed system running DCE.
- It defines time as an interval containing the correct time, rather than as a single value.
- Each node is configured as either a *DTS client* or a *DTS server*
- To synchronize its local clock, a DTS client makes requests to the DTS servers on the same LAN for timing information
- It then uses the method of intersection for computing the new clock value and resets its clock to this value

Distributed Time Service (DTS)

contd…


Time intervals supplied by
Time
DTS server 1
DTS server 2
DTS server 3
Discarded interval
DTS server 4
Largest intersection falling within the remaining intervals
Midpoint of this interval is the new clock value

Distributed Time Service (DTS)

contd…

- Due to the use of the method of intersection for computing new clock values, it is recommended that each LAN has at least three DTS servers.
- The DTS servers of a LAN also communicate among themselves periodically and use the same algorithm to keep their clocks mutually synchronized.

Event Ordering

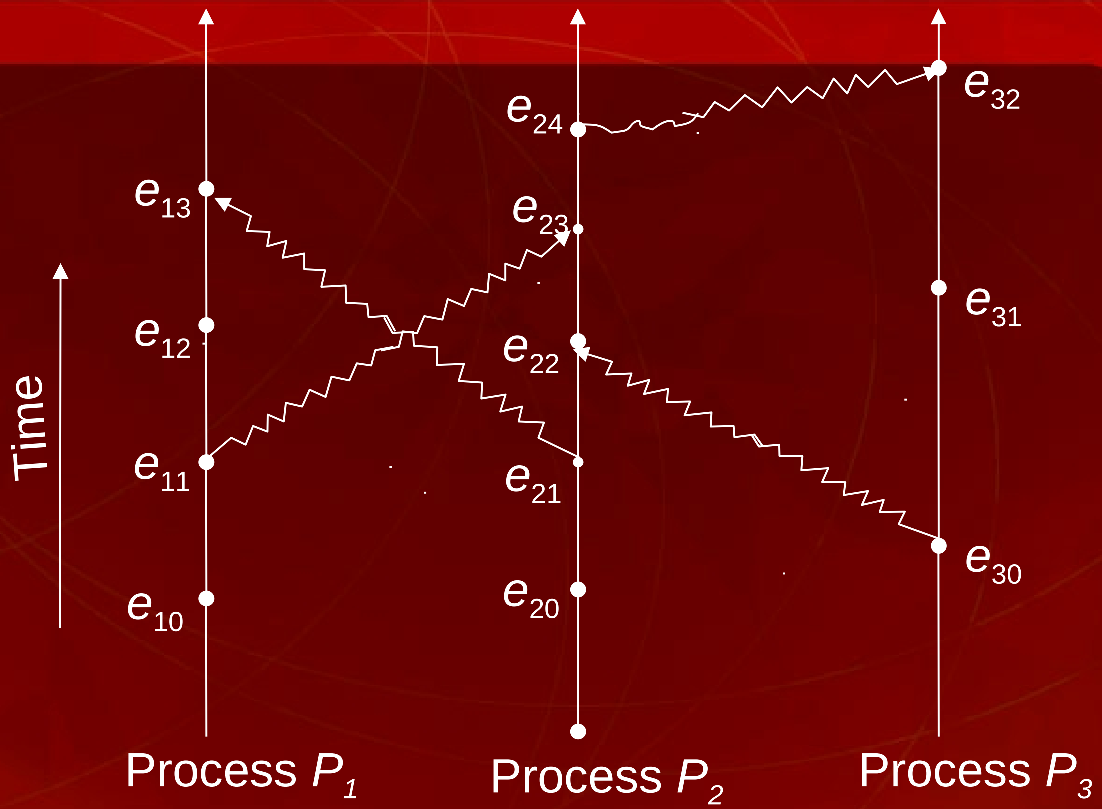
Event Ordering

- For most applications, it is not necessary to keep the clocks in a distributed system synchronized
- It is sufficient to ensure that all events are ordered in a manner that is consistent with an observed behavior
- A new relation (called happened-before relation) along with the concept of logical clocks is used to ensure this.

Happened-Before Relation (Causal Ordering)

- If event ‘a’ occurs before event ‘b’, then $a \rightarrow b$
- If ‘a’ is the event of sending a message and ‘b’ is the event of receiving the same message, then $a \rightarrow b$
- If $a \rightarrow b$ and $b \rightarrow c$, then $a \rightarrow c$
- If two events are not related by the happened before relation, they are said to be concurrent

Illustrating Causal & Concurrent Events



Examples of Causal Events:

$e_{10} \rightarrow e_{11}$; $e_{20} \rightarrow e_{24}$; $e_{11} \rightarrow e_{23}$, $e_{30} \rightarrow e_{24}$, $e_{11} \rightarrow e_{32}$

Examples of Concurrent Events:

(e_{12}, e_{20}); (e_{21}, e_{30}); (e_{10}, e_{30}); (e_{11}, e_{31}); (e_{12}, e_{32}); (e_{13}, e_{22})

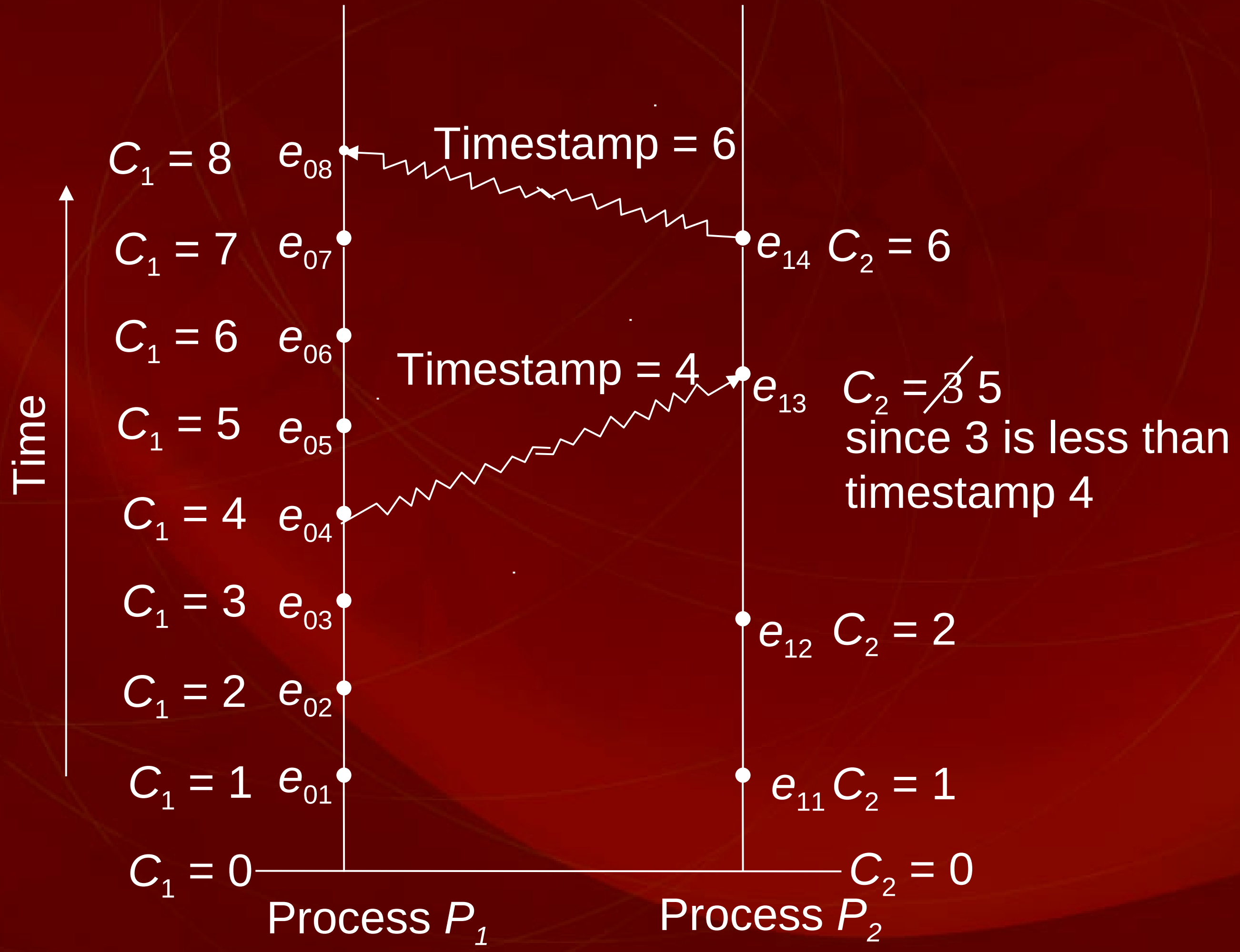
Logical Clocks Concept

- It is a way to properly order events in a system
- It is based on the concept that only those events need to be ordered, which are related to each other by the happened-before relation (directly or indirectly)
- If $a \rightarrow b$, then $c(a) < c(b)$, where $c(a)$ and $c(b)$ are timestamps assigned to events ‘a’ and ‘b’ respectively

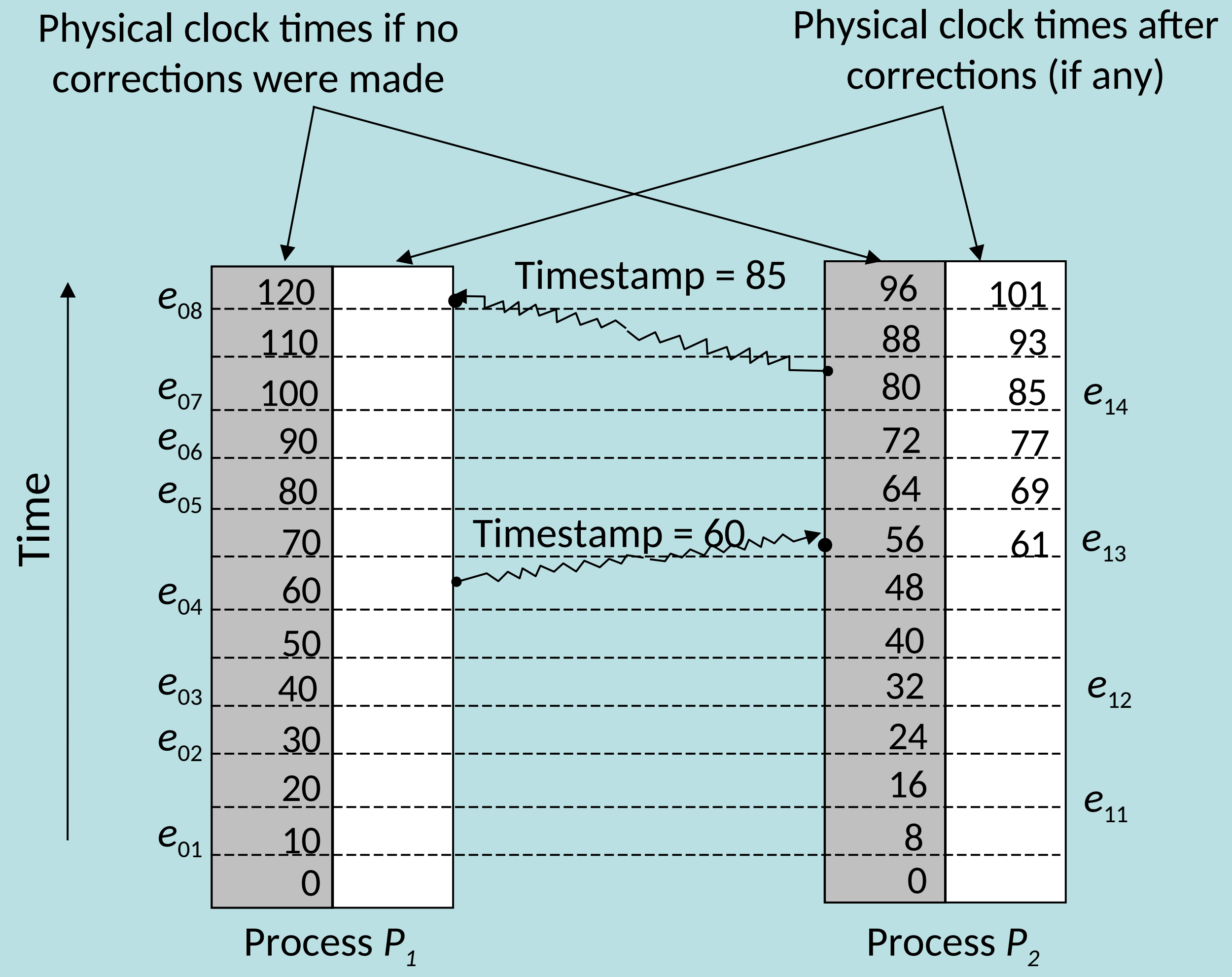
Logical Clocks Implementation

- By using counters
- By using physical clocks

Logical Clocks Implementation by Using Counters



Logical Clocks Implementation by Using Physical Clocks


Physical clock times if no corrections were made
Physical clock times after corrections (if any)
Time
Timestamp = 85
Timestamp = 60
e_{08} e_{07} e_{06} e_{05} e_{04} e_{03} e_{02} e_{01}
e_{14} e_{13} e_{12} e_{11}
120 110 100 90 80 70 60 50 40 30 20 10 0
96 88 80 72 64 56 48 40 32 24 16 8 0
101 93 85 77 69 61
Process P_1
Process P_2

Total Ordering of Events

- The happened-before relation is only a partial ordering on the set of all events in the system
- With this event-ordering scheme, it is possible that two events *a* and *b* that are not directly or indirectly related by the happened-before relation, may have the same timestamps.
- For example, if events *a* and *b* happen respectively in processes P_1 and P_2, when the clocks of both processes show the same time (say 100), both events will have a timestamp of 100.
- Hence, nothing can be said about the order of the two events.

Total Ordering of Events

contd…

- For total ordering of all system events, a mechanism is needed to assign a unique timestamp to each event.
- For this, Lamport proposed the use of any arbitrary total ordering of the processes (such as by using process identity numbers).
- For instance, in the example discussed above, the timestamps associated with events *a* and *b* will be 100.01 and 100.02, respectively, where the process identity numbers of processes P_1 and P_2 are 001 and 002, respectively

Mutual Exclusion

Requirements for a Mutual Exclusion Algorithm

- *Mutual Exclusion*: at any time, only one process should access the resource
- *No Starvation*: Every request for a resource must be eventually granted

Mutual Exclusion in Distributed Systems

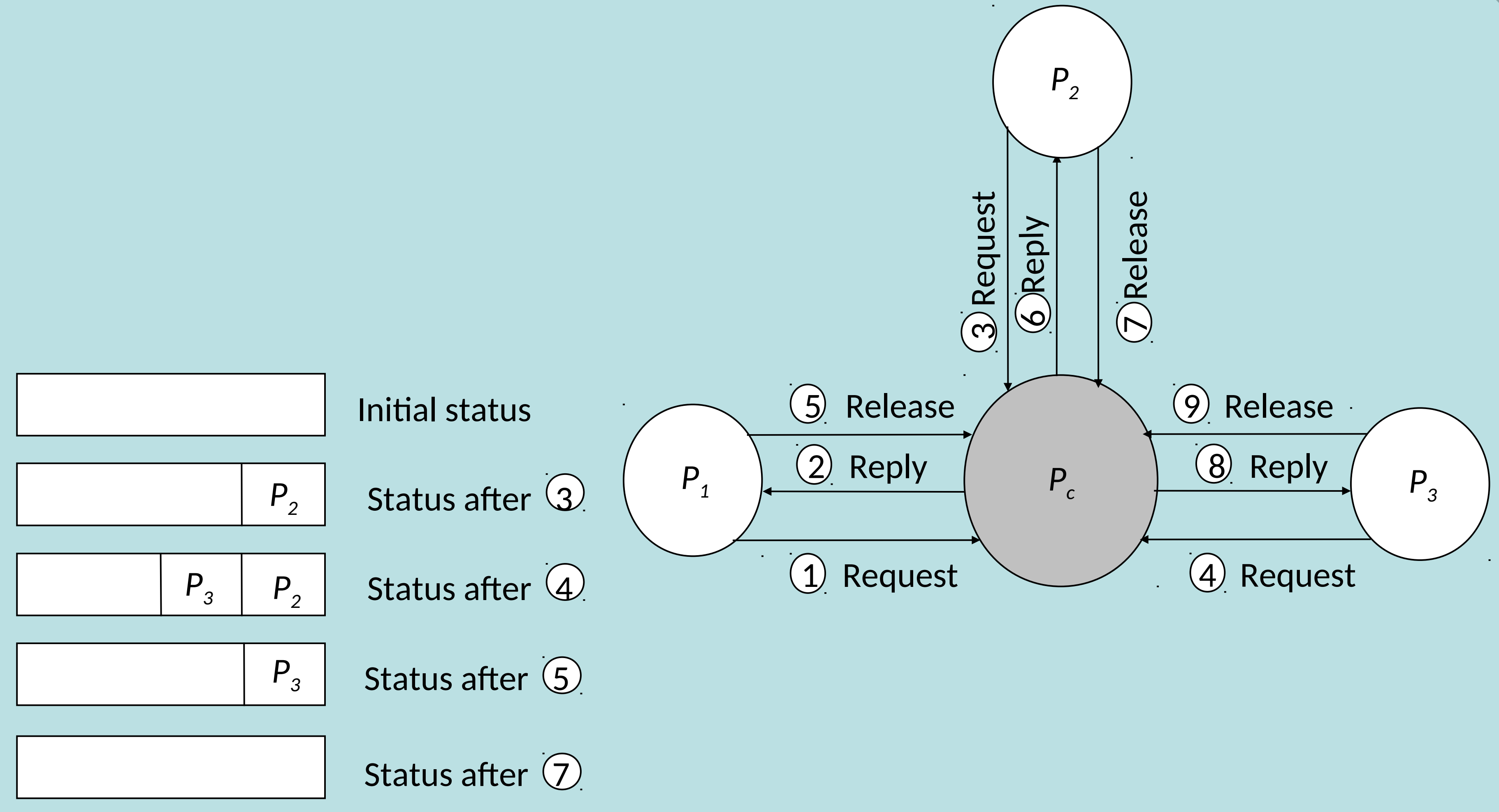
One of the following approaches is used:

- Centralized
- Distributed
- Token passing

Mutual Exclusion – Centralized Approach

- A coordinator process coordinates the entry to the critical sections
- A process wanting to enter a critical section must first seek coordinator’s permission
- The coordinator grants permission to only one process at a time by using some scheduling algorithm
- When a process exits the critical section, it notifies the coordinator

Mutual Exclusion – Centralized Approach



Status of request queue

Mutual Exclusion – Centralized Approach (Discussion)

- It ensures mutual exclusion because the coordinator allows only one process to enter critical section at a time
- It ensures no starvation because of the use of FIFO scheduling policy.
- It is simple to implement
- It requires only three messages per critical section entry – a request, a reply and a release.
- It, however, suffers from the usual drawbacks of centralized schemes – a single point of failure and performance bottleneck in a large system

Mutual Exclusion – Distributed Approach

- All processes that want to enter the same critical section cooperate with each other to decide which process will enter the critical section next.

Mutual Exclusion – Distributed Approach

- One method to do this is as follows:
  - When a process wants to enter a critical section, it sends a request message to all other processes. The message contains the process-id, the critical-section-id and a timestamp generated by the process
  - On receiving this message, a process replies immediately

    IF

    It neither is in the critical section nor is waiting for its turn to enter the critical section

    OR

    If the timestamp of the received request message is lower than the timestamp of its own request message

    ELSE

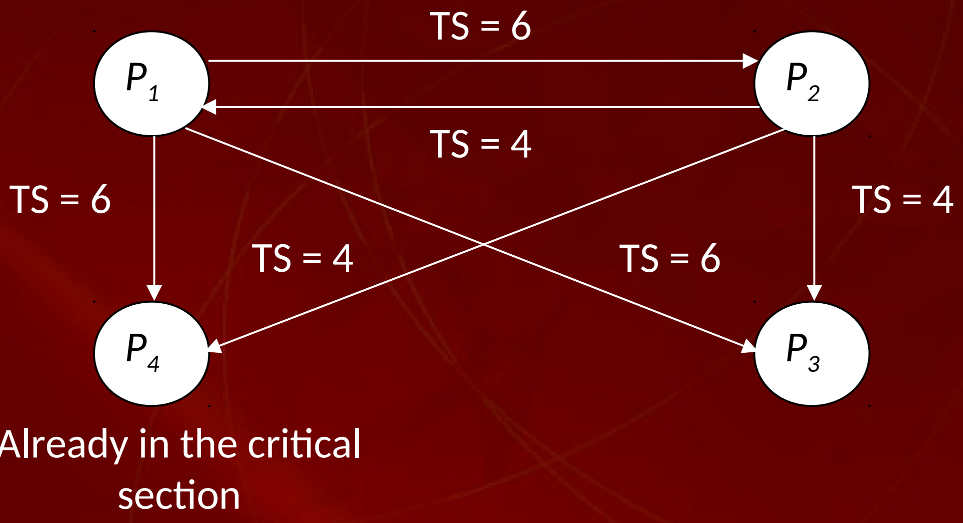
    It queues the request message and defers sending a reply

Mutual Exclusion – Distributed Approach

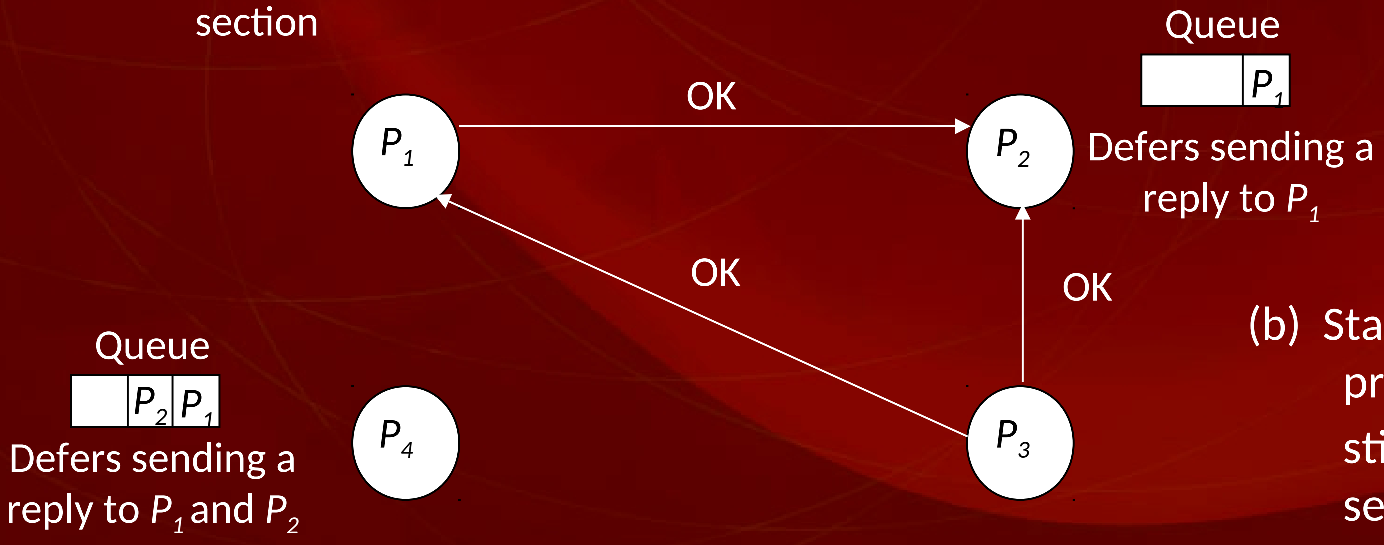
(contd…)

- A process enters the critical section as soon as it has received reply messages from all processes
- After exiting from the critical section, it sends reply messages to all processes in its queue and deletes them from its queue

Mutual Exclusion – Distributed Approach

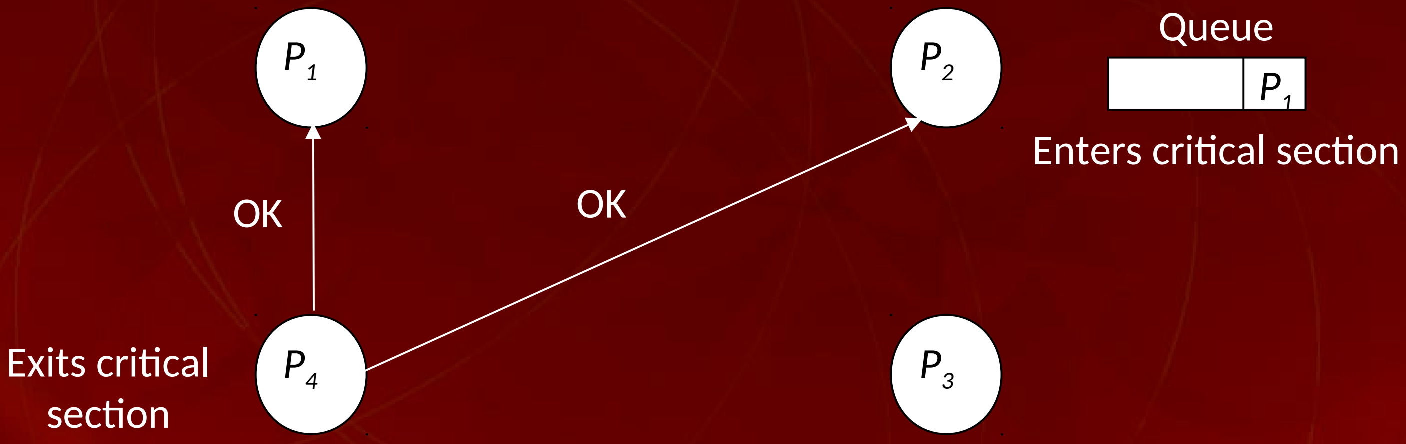


(a) Status when processes P_1 and P_2 send request messages to other processes while process P_4 is already in the critical section

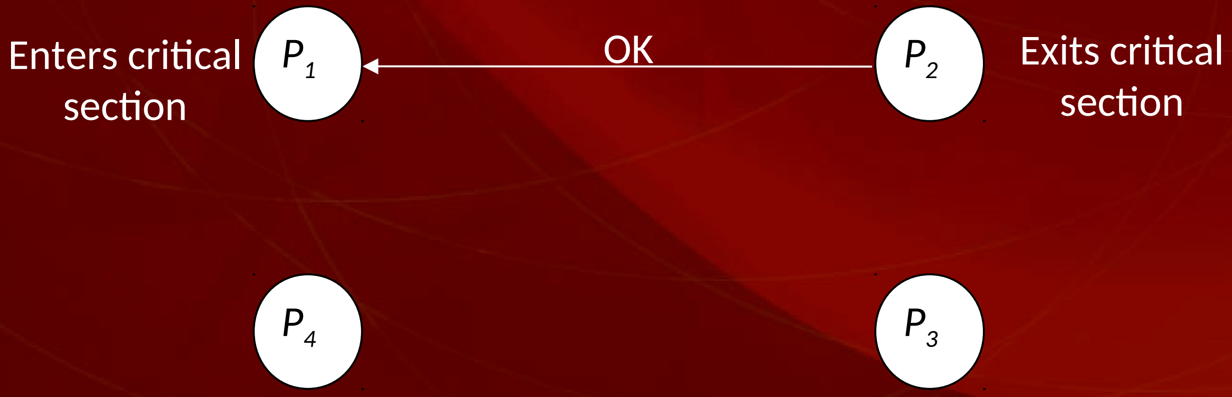

Queue
P_1
OK
P_1
P_2
Defers sending a reply to P_1
OK
OK
Queue
P_2 P_1
Defers sending a reply to P_1 and P_2
P_4
P_3

(b) Status while process P_4 is still in critical section

Mutual Exclusion – Distributed Approach



(c) Status after process P_4 exits critical section



(d) Status after process P_2 exits critical section

Mutual Exclusion – Distributed Approach (Discussion)

- It ensures mutual exclusion because a processes can enter its critical section only after getting permission from all other processes.
- It ensures no starvation since entry to the critical section is scheduled according to the timestamp ordering.
- It has been proved that the algorithm is free from deadlock.
- For *n* processes, the algorithm requires *(n-1)* request messages and *(n-1)* reply messages, giving a total of *2(n-1)* messages per critical section entry. Hence, it is suitable only for a small group of cooperating processes

Mutual Exclusion – Distributed Approach (Discussion)

- For *n* processes, the algorithm is liable to *n* points of failure because if one of the processes fails, the entire scheme collapses.
- A simple method to solve the above problem is to send a reply message (either “permission denied” or OK) immediately to the requesting process rather than keeping quiet till it is time to send the OK message when the permission can be granted. Another method is to use majority consensus rather than the consensus of all processes.
- It requires each process to know the identity of the participating processes. This makes its implementation complex because each process of a group needs to dynamically keep track of the processes entering/leaving the group.

Mutual Exclusion – Token Passing Approach

- A *token* is a special message that entitles its holder to enter a critical section.
- The processes are organized in a logical ring, and the token is circulated around the ring.
- When a process receives the token

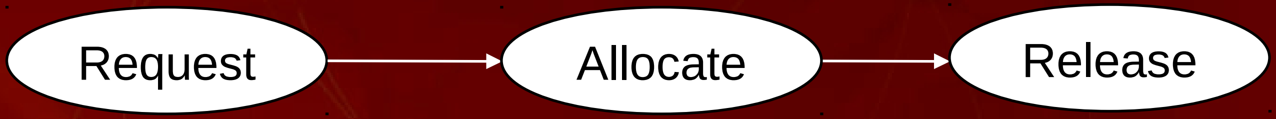
```
    IF
        It does not want to enter the critical section, it
        passes it to its neighbor
    ELSE
        It keeps the token, executes its critical
        section, and then passes it to its neighbor
```

Mutual Exclusion – Token Passing Approach (Discussion)

- It ensures mutual exclusion because at any time, only one process can be in critical section, since there is only one token.
- It ensures no starvation, since the ring is unidirectional and a process is permitted to enter only one critical section each time it gets the token.
- The number of messages per critical section entry may vary from one (when every process always wants to enter a critical section) to an unbounded value (when no process wants to enter a critical section).
- For *n* processes, the waiting time from the moment a process wants to enter a critical section until its actual entry may vary from the time needed to exchange 0 to *(n-1)* token-passing messages.

Deadlock

Resource Utilization Sequence



Allocate is the only operation that the system can control

Deadlock Definition

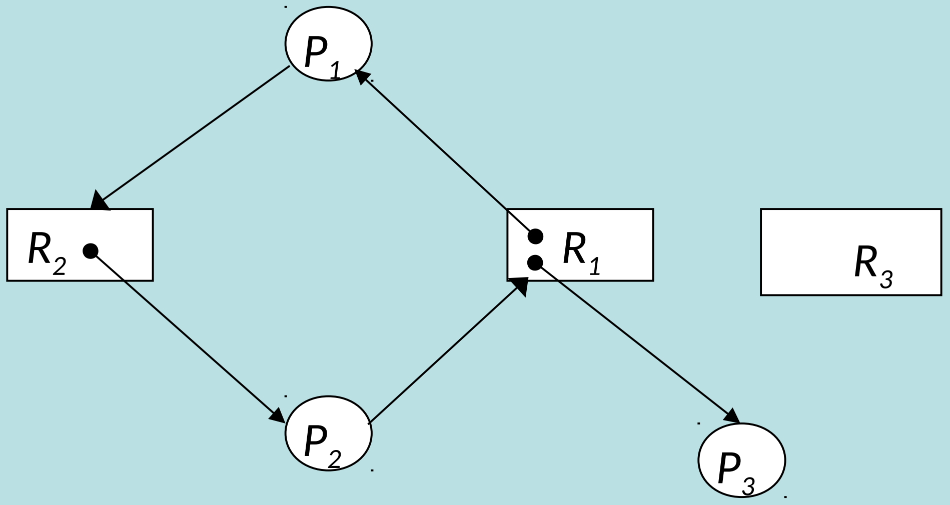
Deadlock is the state of permanent blocking of a set of processes each of which is waiting for an event that only another process in the set can cause

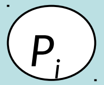
Necessary Conditions for Deadlock

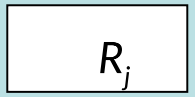
- Mutual-exclusion condition
- Hold-and-wait condition
- No-preemption condition
- Circular-wait condition

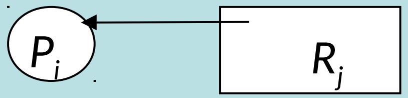
All four conditions must hold simultaneously in a system for deadlock to occur

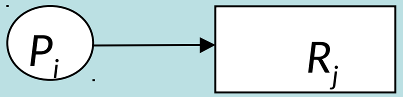
Deadlock Modeling – Resource Allocation Graph




A process named P_i


A resource R_j having 3 units in the system

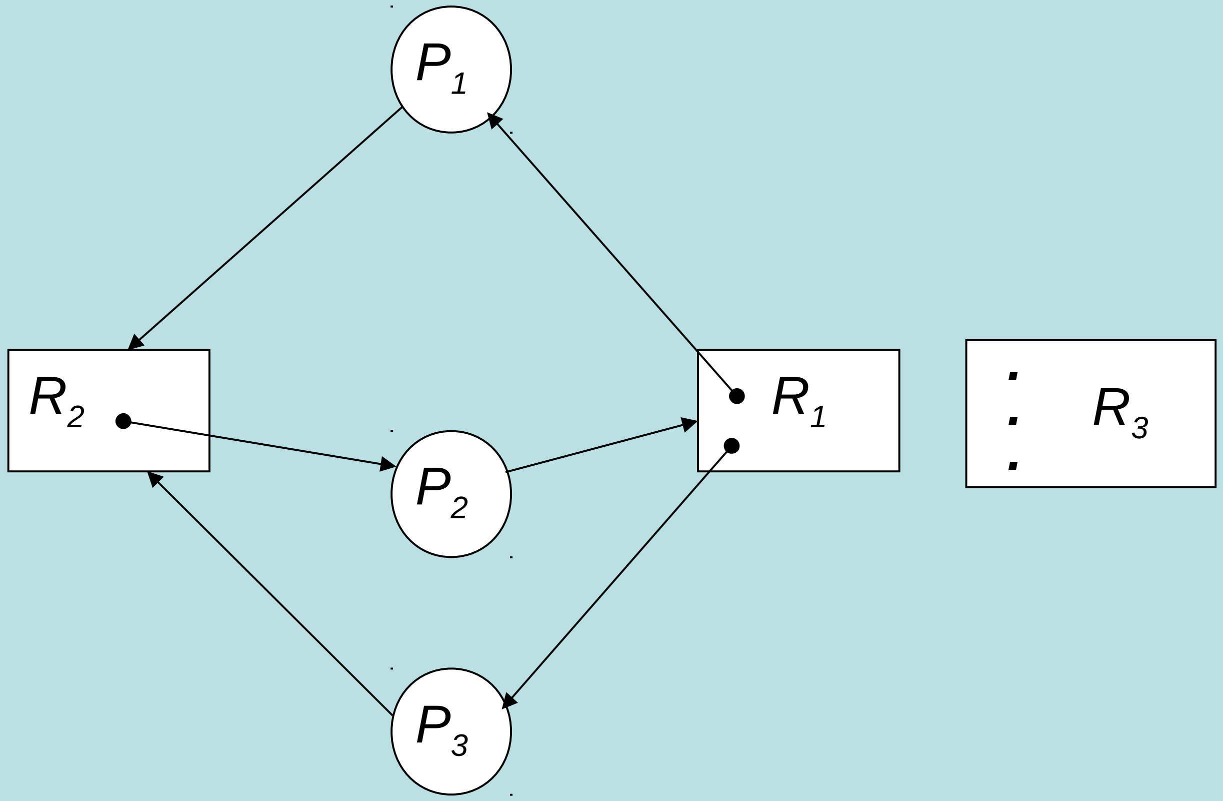

Process P_i holding a unit of resource R_j


Process P_i requesting for a unit of resource R_j

Necessary & Sufficient Conditions for Deadlock

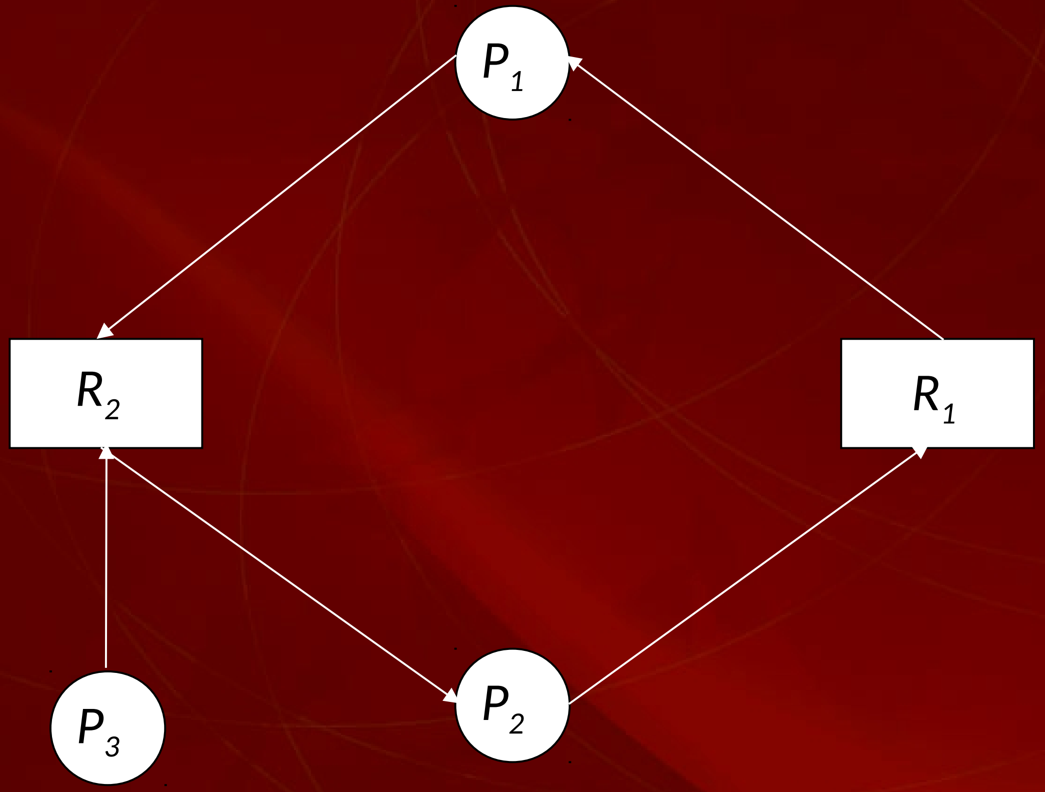
- A cycle is a necessary condition.
- If there is only a single unit of each resource type involved in the cycle, a cycle is both a necessary and a sufficient condition.
- If one or more of the resource types involved in the cycle have more than one unit, a knot is a sufficient condition.

An Example of a Knot

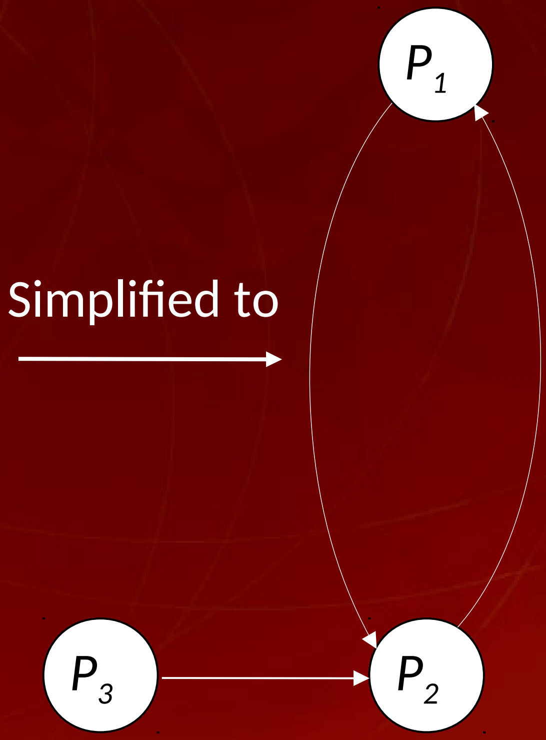


A knot is a nonempty set *K* of nodes such that the reachable set of each node in *K* is exactly the set *K*. A knot always contains one or more cycles.

Wait-for Graph from Resource Allocation Graph



(*a*) Resource allocation graph



(*b*) Corresponding WFG

Methods for Handling Deadlocks

The three commonly used strategies are:

- Avoidance
- Prevention
- Detection and recovery

Deadlock Avoidance

- Safe and unsafe state analysis is done before allocating a resource to a requesting process and resource allocation is done in such a manner to ensure that the system will always remain in a safe state.
- Since the initial state of a system is always a safe state, whenever a process requests a resource that is currently available, the system checks to find out if this allocation will change the system state from safe to unsafe.
- If no, the request is immediately granted, else it is deferred.

Deadlock Avoidance

	P_1	P_2	P_3
Max	4	5	6
Holds	2	2	2

(a) Free = 2

	P_1	P_2	P_3
Max	4	5	6
Holds	4	2	2

(b) Free = 0

	P_1	P_2	P_3
Max	—	5	6
Holds	0	2	2

(c) Free = 4

	P_1	P_2	P_3
Max	—	5	6
Holds	0	5	2

(d) Free = 1

	P_1	P_2	P_3
Max	—	—	6
Holds	0	0	2

(e) Free = 6

	P_1	P_2	P_3
Max	—	5	6
Holds	0	2	6

(f) Free = 0

	P_1	P_2	P_3
Max	—	5	—
Holds	0	2	0

(g) Free = 6

Deadlock Avoidance

(contd…)

	P_1	P_2	P_3
Max	4	5	6
Holds	2	2	2

(a) Free = 2

	P_1	P_2	P_3
Max	4	5	6
Holds	2	3	2

(b) Free = 1

Demonstration that an allocation may move the system from a safe to an unsafe state

Deadlock Avoidance - Discussion

Although theoretically attractive, this method is rarely used in practice due to the following reasons:

- It requires advance knowledge of the resource requirements of the various processes.
- It assumes that the number of processes that compete for a particular resource is fixed and known in advance.
- It restricts resource allocation too severely and consequently degrades the system performance.

The above limitations become more sever in a distributed system because the collection of information needed for making resource allocation decisions at one point is difficult and inefficient.

Deadlock Prevention Methods

- Collective requests
  - denies hold-and-wait condition
- Ordered requests
  - denies circular-wait condition
- Preemption
  - denies no-preemption condition
- Spooling of outputs sent to a printer onto a disk
  - denies mutual-exclusion condition

Deadlock Detection & Recovery

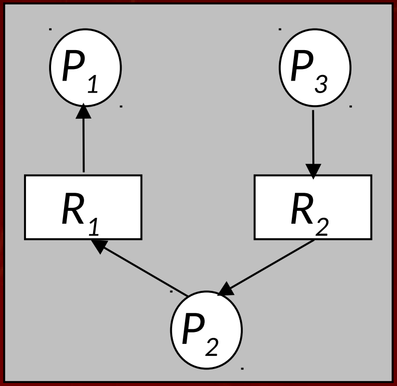
Assuming a single unit of each resource type, the method involves:

- Maintaining WFG
- Searching for cycles in the WFG
- If a cycle (deadlock) is detected, using one of the following methods to recover:
  - Asking for operator intervention
  - Termination of process(es)
  - Rollback of process(es)

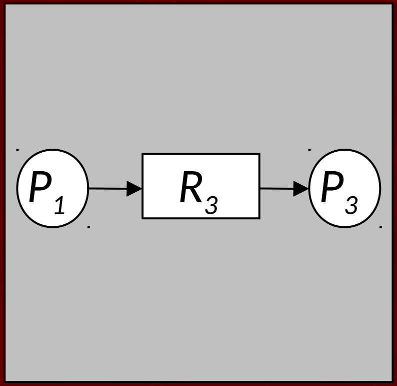
Construction of the WFG in a Distributed System

1. Construct a resource allocation graph separately for each site.
2. Convert the resource allocation graphs to their corresponding WFGs
3. Take the union of the WFGs of all sites and construct a single global WFG

Construction of the WFG in a Distributed System

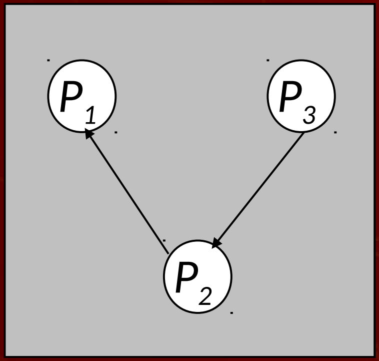


Site S_1

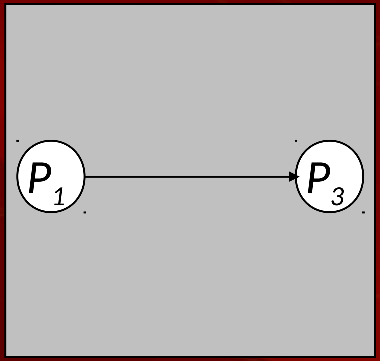


Site S_2

(a) Resource allocation graphs of each site.

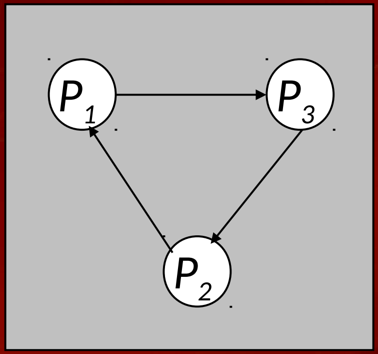


Site S_1



Site S_2

(b) WFGs corresponding to graphs in (*a*)



(c) Global WFG by taking the union of the two local WFGs of (*b*)

Construction of WFG in a Distributed System

The three commonly used techniques for organizing the WFG in a distributed system are:

- Centralized
- Hierarchical
- Distributed

Centralized Approach for Deadlock Detection

- A local coordinator at each site maintains a WGF for its local resources.
- A central coordinator constructs the union of all the individual WFGs.
- A cycle in a local WFG, represents a local deadlock, and resolved locally by the local coordinator of the site.
- Deadlocks involving resources at two or more sites get reflected as cycles in the global WFG, and resolved by the central coordinator.

Centralized Approach for Deadlock Detection

(contd…)

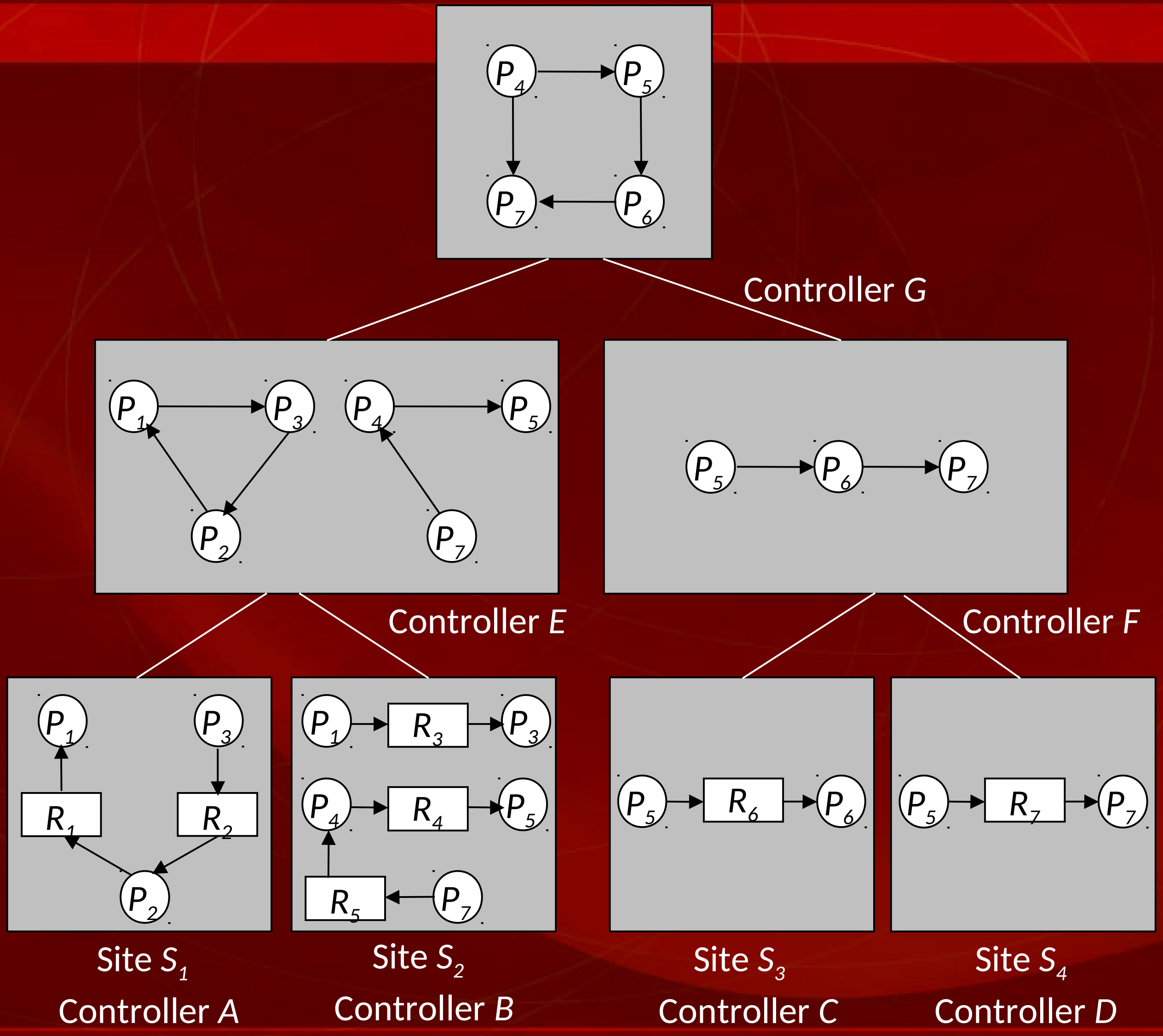
Transfer of WFG-related information from local coordinators to the central coordinator is done in one of the following ways:

- *Continuous Transfer*: A message is sent whenever a new edge is added/deleted to a local WFG.
- *Periodic Transfer*: A message is sent periodically, which may include one or more changes to a local WFG.
- *Transfer-on-Request*: The central coordinator invokes the cycle detection algorithm periodically and requests information from each site just before invoking the algorithm.

Hierarchical Approach for Deadlock Detection

- Uses a logical hierarchy (tree) of deadlock detectors (called controllers)
- Each controller forming a leaf of the tree maintains the local WFG of a single site
- Each non-leaf controller maintains a WFG that is the union of the WFGs of its immediate children in the tree
- The lowest level controller that finds a cycle in its WFG detects a deadlock and takes necessary action to resolve it

Hierarchical Approach for Deadlock Detection



Fully Distributed Approaches for Deadlock Detection

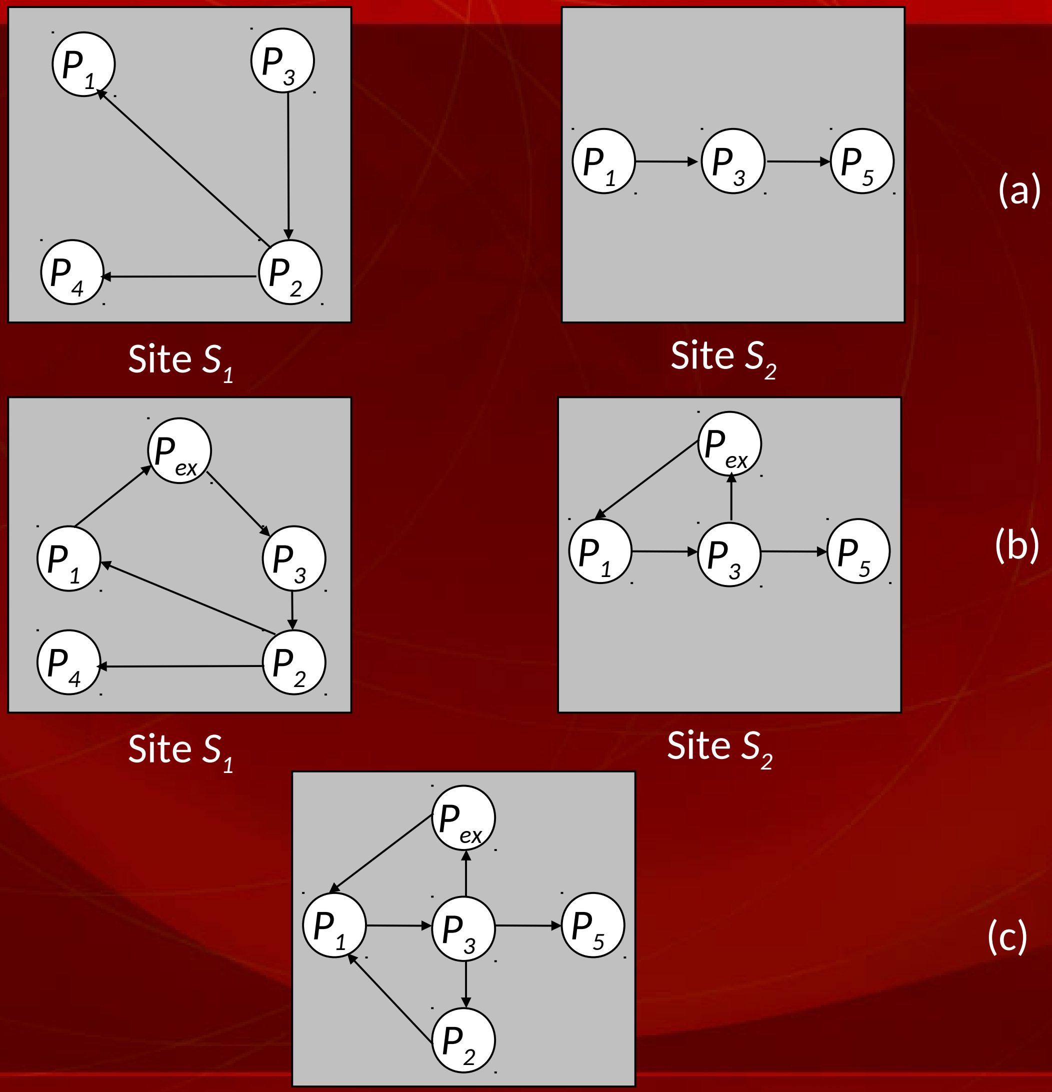
Two such algorithms are:

- WFG-based
- Probe-based

WFG-Based Distributed Algorithm for Deadlock Detection

- Each site maintains its own local WFG
- To model waiting situations that involve non-local processes, an extra node (P_{ex}) is added to each local WFG in the following manner:
  - An edge (P_i, P_{ex}) is added if process Pi is waiting for a resource in another site being held by any process
  - An edge (P_{ex}, P_j) is added if P_j is a process of another site that is waiting for a resource currently being held by a process of this site

WFG-Based Distributed Algorithm for Deadlock Detection



Example illustrating the WFG-based fully distributed deadlock detection algorithm: *(a)* Local WFGs*(b)* Local WFGs after addition of node P_{ex} *(c)* Updated local WFG of site S_2 after receiving the deadlock detection message from site S_1

WFG-Based Distributed Algorithm for Deadlock Detection

The modified WFGs are used for deadlock detection as follows:

- A cycle in a local WFG that does not involve P_{ex} indicates a local deadlock, and resolved locally.

WFG-Based Distributed Algorithm for Deadlock Detection

- A cycle in a local WFG that involves P_{ex} (P_{ex}, P_i, P_j, …, P_k, P_{ex}) indicates a possibility of a distributed deadlock, which is confirmed as follows:
  - S_i sends a message to S_j containing that part of WFG that forms the cycle
  - S_j updates its local WFG by adding those edges of the cycle that do not involve node P_{ex} to its WFG
  - If the newly constructed WFG of site S_j contains a cycle that does not involve node P_{ex}, a deadlock is confirmed
  - Else, S_j sends a message to the appropriate site (say S_k), and the whole process is repeated by site S_k

Probe-Based Distributed Algorithm for Deadlock Detection

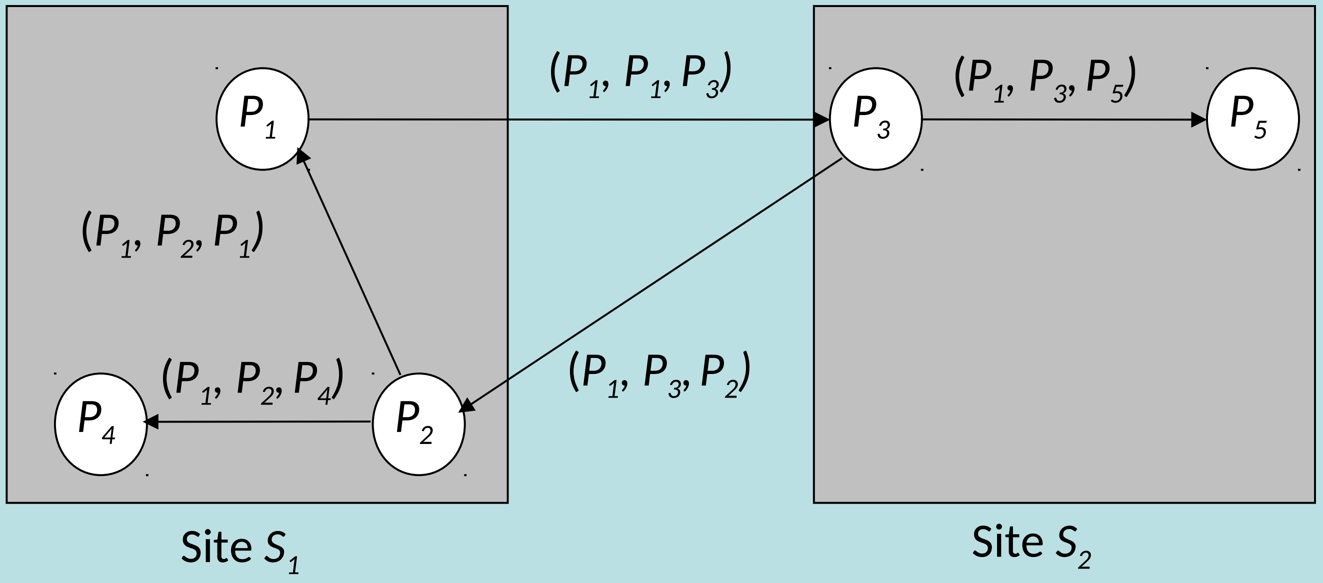
- When a process fails to get a requested resource and times out, it generates a probe message and sends it to the process holding the resource. The message contains the following:
  1. ID of the process just blocked
  2. ID of the process sending this message
  3. ID of the process to whom this message is being sent
- On receiving this message, if the recipient process finds that it is using the resource, it ignores the message.

Probe-Based Distributed Algorithm for Deadlock Detection

(contd…)

- Else, if the recipient is itself waiting for the resource, it passes the probe message to the process holding the resource. The message is modified as follows:
    1. The first field is unchanged
    2. The 2nd field is changed to the recipient's ID
    3. The 3rd field is changed to the ID of the new process to whom this message is being sent
- This process gets repeated, and if the probe message returns back to the original sender, a cycle exists and the system is deadlocked

Probe-Based Distributed Algorithm for Deadlock Detection



Probe-Based Distributed Algorithm is the Best Known Algorithm Because

1. The algorithm is easy to implement, since each message is of fixed length and requires few computational steps.
2. The overhead of the algorithm is fairly low.
3. There is no graph constructing and information collecting involved.
4. False deadlocks are not detected by the algorithm.
5. It does not require any particular structure among the processes.

Election Algorithms

Election Algorithms

- A coordinator process is often needed in a distributed system (for example, such a process is needed in the centralized algorithm for mutual exclusion and the central coordinator in the centralized deadlock detection algorithm)
- All concerned processes must agree to who the coordinator process is
- Election algorithms are meant for electing a coordinator process from among the currently running processes
- If the coordinator process fails, a new coordinator process must be elected

Assumptions for Election Algorithms

- Each process in the system has a unique priority number
- Whenever an election is held, the process having the highest priority among the currently active processes is elected as the coordinator
- On recovery, a failed process can take appropriate action to rejoin the set of active processes

Therefore, whenever initiated , an election algorithm finds out the process having highest priority number among the active processes, and then informs this to all other active processes.

Commonly Used Election Algorithms

- Bully Algorithm
- Ring Algorithm

Bully Algorithm for Election

- If a process (P_i) sends a *request* message to the coordinator and times out, it assumes that the coordinator has failed.
- It then initiates an election by sending an *election* message all processes with a higher priority number than itself
- If P_i does not receive any response, it declares itself as the coordinator by sending a *coordinator* message to all processes with a lower priority number than itself.
- If P_i receives a response (some other process with a higher priority is alive), it just waits to receive the final results (a coordinator message) of the election it initiated.
- A process which responds also takes over the election activity and repeats the process. Hence, the active process with the highest priority number always wins the election.

Ring Algorithm for Election

- All the processes are organized in a logical, unidirectional ring.
- If a process (P_i) sends a request message to the coordinator and times out, it assumes that the coordinator has failed
- It then initiates an election by sending an election message to its neighbor. It includes its priority number in the message.
- On receiving the message, the recipient appends its own priority number to the message and forwards it to its own neighbor. The process continues.
- When the message comes back to P_i, it selects the highest priority number from the list and informs this to all processes by sending a *coordinator* message around the ring.

Discussion of the Two Election Algorithms

The ring algorithm is more efficient and easier to implement than the bully algorithm because:

- For holding an election, if *n* is the total number of processes, bully algorithm requires $O(n^2)$ messages in the worst case (when the process having the lowest priority initiates an election), and only $(n-2)$ messages in the best case (when the process having the highest priority initiates an election)
- In the ring algorithm, only $2(n-1)$ messages are required for an election, irrespective of which process initiates an election.

Discussion of the Two Election Algorithms

contd...

- In bully algorithm, recovery of a process involves initiation of an election by the process. Hence, once again the bully algorithm requires *$O(n^2)$* messages in the worst case, and *$(n-1)$* messages in the best case, depending on the priority of the process that initiates the recovery action.
- In ring algorithm, a failed process does not initiate an election but simply searches for the current coordinator. Hence, it requires only *$n/2$* messages on an average for recovery action.

Thank you

Dr Pradeep K Sinha

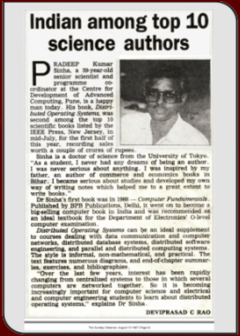
Distinguished Engineer

Association for Computing Machinery

Fellow

Institute of Electrical and Electronics Engineers

"Envisioning the Future of India by Grooming Young Minds"



Indian among top 10 science authors

PRADEEP Kumar Sinha, a 39-year-old senior scientist and programme co-ordinator at the Centre for Development of Advanced Computing, Pune, is a happy man today. His book, *Distributed Operating Systems*, was second among the top 10 scientific books listed by the IEEE Press, New Jersey, in mid-July, for the first half of this year, recording sales worth a couple of crores of rupees.

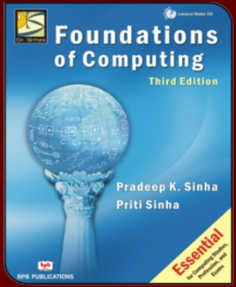
Sinha is a doctor of science from the University of Tokyo. "As a student, I never had any dreams of being an author. I was never serious about anything. I was inspired by my father, an author of commerce and economics books in Bihar. I became serious about studies and developed my own way of writing notes which helped me to a great extent to write books."

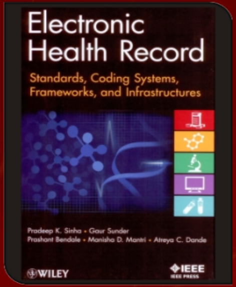
Dr Sinha's first book was in 1988 — *Computer Fundamentals*. Published by BPB Publications, Delhi, it went on to become a big-selling computer book in India and was recommended as an ideal textbook for the Department of Electronics' O-level computer examination.

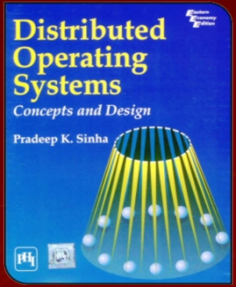
Distributed Operating Systems can be an ideal supplement to courses dealing with data communication and computer networks, distributed database systems, distributed software engineering, and parallel and distributed computing systems. The style is informal, non-mathematical, and practical. The text features numerous diagrams, and end-of-chapter summaries, exercises, and bibliographies.

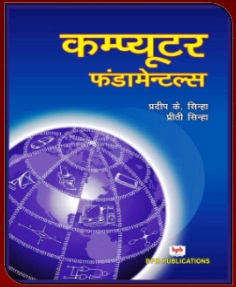
"Over the last few years, interest has been rapidly changing from centralized systems to those in which several computers are networked together. So it is becoming increasingly important for computer science and electrical and computer engineering students to learn about distributed operating systems," explains Dr Sinha.

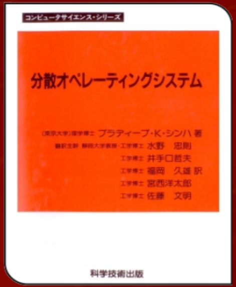
DEVIPRASAD C RAO




Electronic Health Record
Standards, Coding Systems, Frameworks, and Infrastructures
Pradeep K. Sinha · Gaur Sunder
Prashant Bendale · Manisha D. Mantri · Atreya C. Dande
WILEY
IEEE PRESS






コンピュータサイエンス・シリーズ
分散オペレーティングシステム
(東京大学)理学博士 プラディープ・K・シンハ 著
水野 忠則
井手口哲夫
福岡 久雄 訳
宮西洋太郎
佐藤 文明
科学技術出版