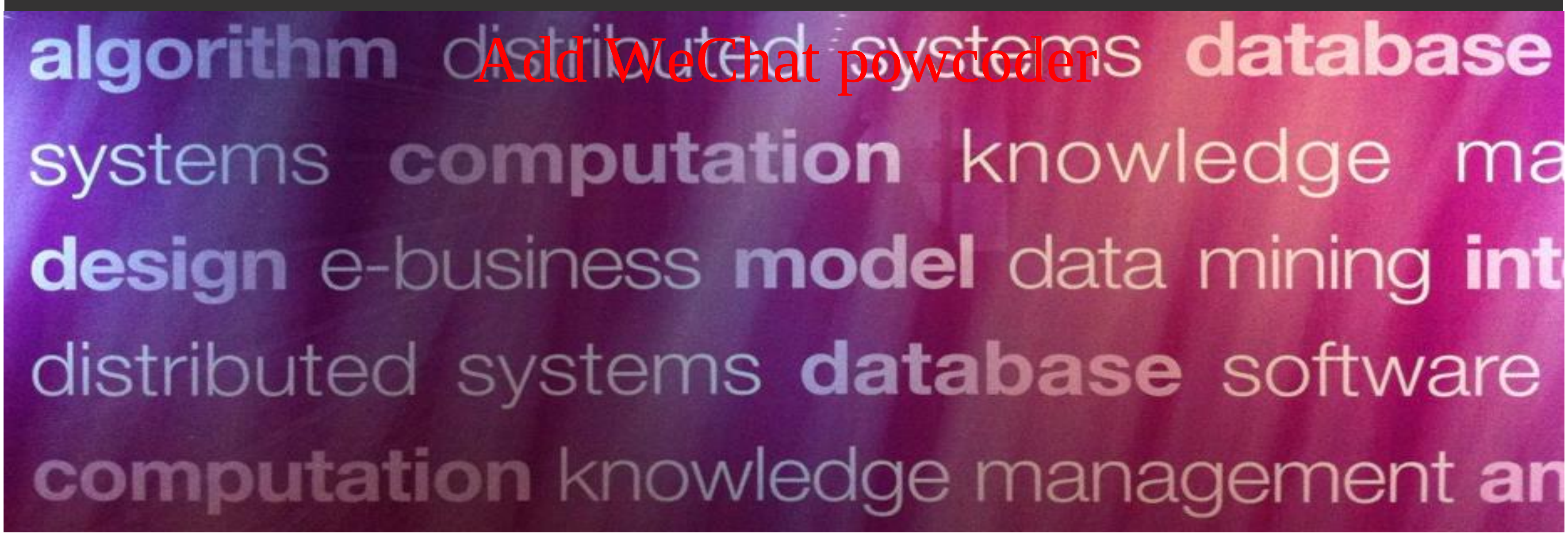
MONASH University

Information Technology

FIT3143 - LECTURE WEEK 6

SYNCHRONIZATION, MUTEX, DEADLOCKS



Overview

- Real time Clock Synchronization Methods
- Logical Clock Synchronization Techniques
- Mutual Exclusion Approaches
- Deadlock detection and handling

Learning outcome(s) related to this topic

- Explain the fundamental principles of parallel computing architectures and algorithms (LO1)
- Compare and contrast different parallel computing architectures, algorithms and communication schemes using research-based knowledge and methods (LO2)

Synchronisation in Distributed Systems

- A Distributed System consists of a collection of distinct processes that are spatially separated and run concurrently
- In systems with multiple concurrent processes, it is economical to share the system resources.

- Sharing may be cooperative or competitive
- Both competitive and cooperative sharing require adherence to certain rules of behavior that guarantee that correct interaction occurs.
- The rules of enforcing correct interaction are implemented in the form of synchronization mechanisms.

Issues implementing synchronization in DS

- In single CPU systems, synchronization problems such as mutual exclusion can be solved using semaphores and monitors. These methods rely on the existence of shared memory.

- We cannot use semaphores and monitors in distributed systems since two processes running on different machines cannot expect to have shared memory.

- Even simple matters such as determining one event happened before the other event requires careful thought.

Issues implementing synchronization in DS

- In distributed systems, it is usually not possible or desirable to collect all the information about the system in one place and synchronization among processes is difficult due to the following features of distributed systems:
  - The relevant information is scattered among multiple machines.
  - Processes make decisions based only on local information.
  - A single point of failure in the system should be avoided.
  - No common clock or other precise global time source exists.

Time and Distribution: Why?

- External reasons: We often want to measure time accurately
  - For billings: How long was computer X used?
  - For legal reasons: When was credit card W charged?
  - For traceability: When did this attack occurred? Who did it?
  - – System must be in sync with an external time reference
    - Usually the world time reference: UTC (Coordinated Universal Time)

- Internal reasons: many distributed algorithms use time
  - Kerberos (authentication server) uses time-stamps
  - This can be used to serialise transactions in databases
  - This can be used to minimise updates when replicating data
  - – System must be in sync internally
    - No need to be synchronised on an external time reference

Clock Synchronization

- Time is unambiguous in a centralized system.

- A process can just make a system call to know the time.
  - If process *A* asks for the current time, and a little later process *B* asks for the time, the value of *B_time* > *A_time*.

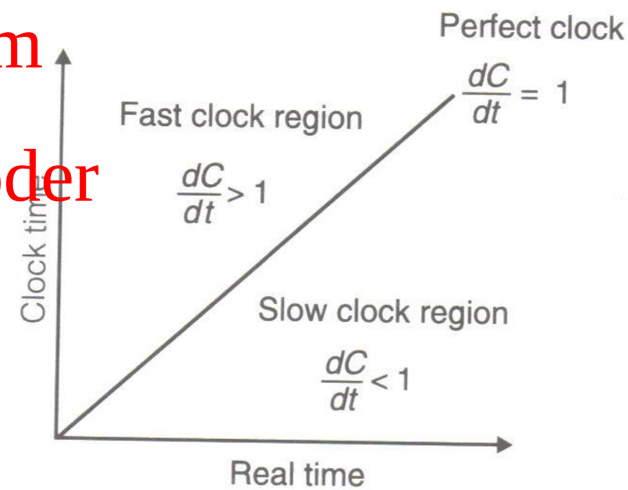
- In a distributed system, if process *A* and *B* are on different machines, *B_time* may not be greater than A_time.

Imperfect Clocks

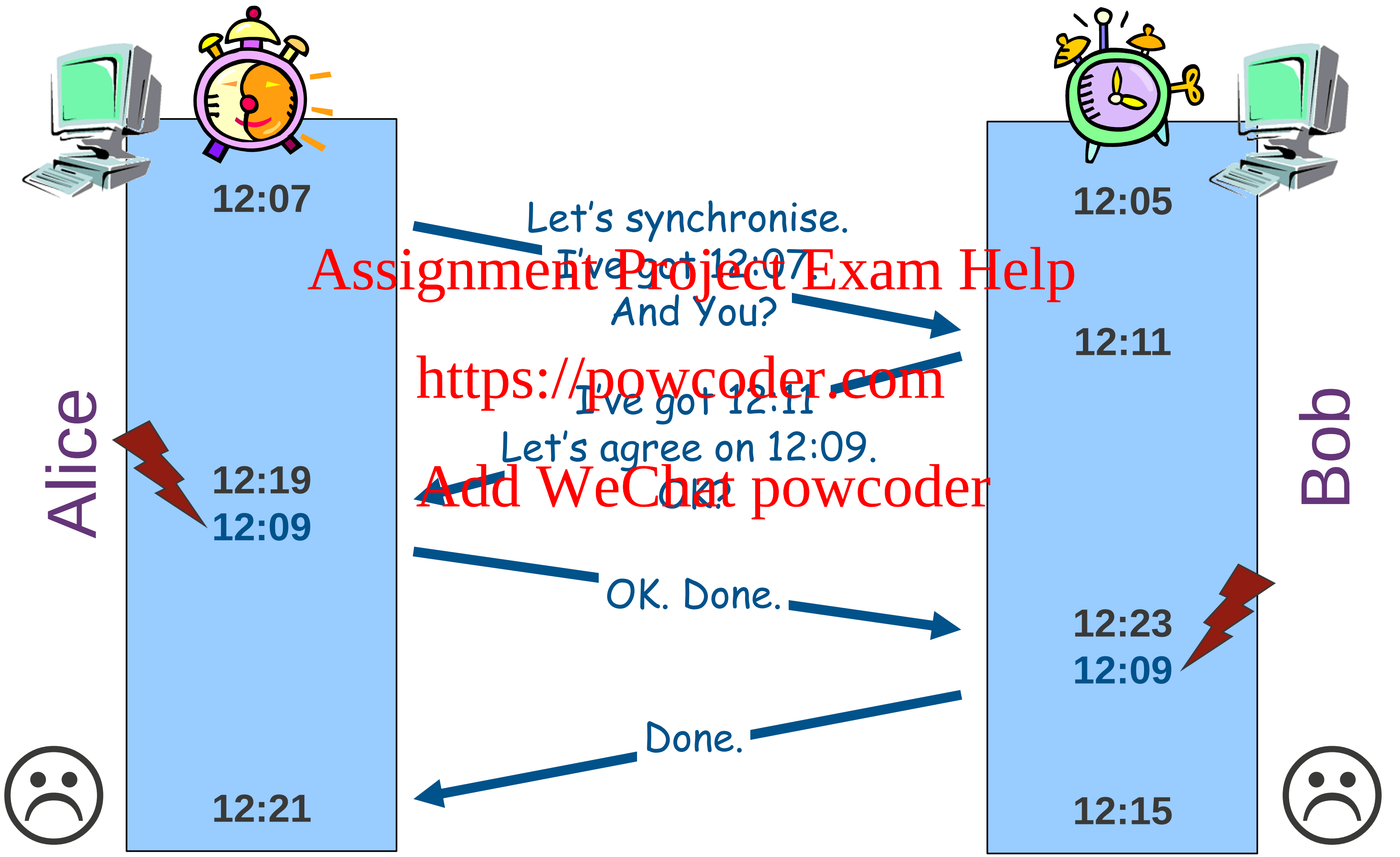
- Human-made clocks are imperfect
  - They run slower or faster than “real” physical time
  - How much faster or slower is called the drift
  - A drift of 1% (i.e. $1/100=10^{-2}$) means the clock adds or looses a second every 100 seconds

- Suppose, when the real time is t the time value of a clock p is $C_p(t)$. If the maximum drift rate allowable is ρ, a clock is said to be non-faulty if the following condition holds -

$$1-\rho \leq \frac{dC}{dt} \leq 1+\rho$$



Clock Synchronisation

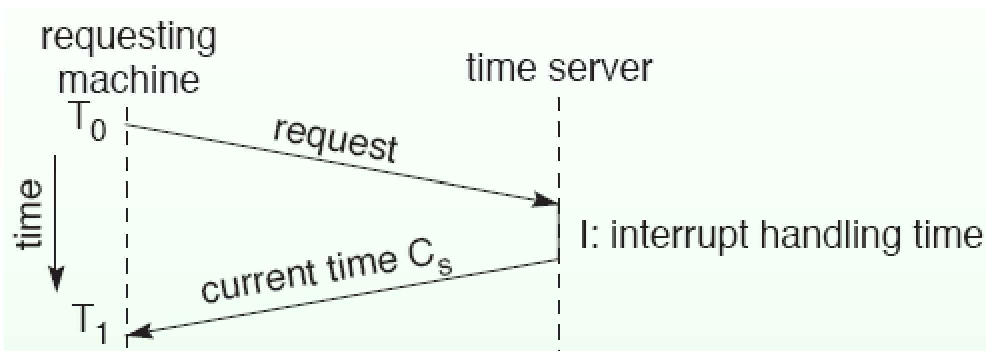


Cristian's Algorithm

- This algorithm synchronizes clocks of all other machines to the clock of **one** machine**, time server**.
- If the clock of the time server is adjusted to the real time, all the other machines are synchronized to the real time.

- Every machine requests the current time to the time server.

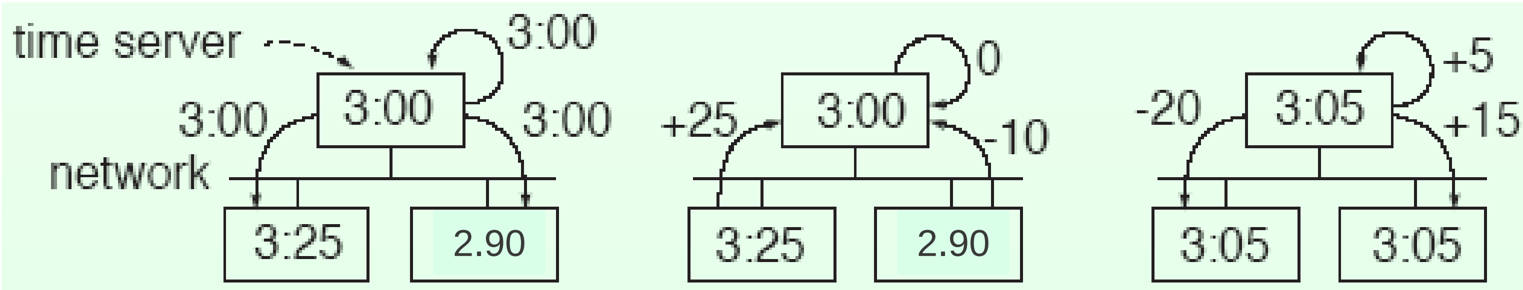
- The time server responds to the request as soon as possible.
- The requesting machine sets its clock to $C_s + (T1 - T0 - I)/2$. In order to avoid clocks moving backward, clock adjustment must be introduced gradually.



The Berkeley Algorithm

- Developed by Gusella and Zatti.
- Unlike Cristian's Algorithm the server process in Berkeley algorithm, called the *master* periodically polls other *slave* process.
- Generally speaking the algorithm is as follows:
  - A *master* is chosen with a ring based election algorithm (Chang and Roberts algorithm).
  - The *master* polls the *slaves* who reply with their time in a similar way to Cristian's algorithm
  - The *master* observes the round-trip time (RTT) of the messages and estimates the time of each *slave* and its own.
  - The *master* then averages the clock times, ignoring any values it receives far outside the values of the others.
  - Instead of sending the updated current time back to the other process, the *master* then sends out the amount (positive or negative) that each *slave* must adjust its clock. This avoids further uncertainty due to RTT at the *slave* processes.
  - Everybody adjust the time.

The Berkeley Algorithm



- With this method the average cancels out individual clock's tendencies to drift.

- Computer systems normally avoid rewinding their clock when they receive a negative clock alteration from the master. Doing so would break the property of monotonic time, which is a fundamental assumption in certain algorithms in the system itself or in some programs.

- A simple solution to this problem is to halt the clock for the duration specified by the master, but this simplistic solution can also cause problems, although they are less severe. For minor corrections, most systems slow the clock (known as "clock slew"), applying the correction over a longer period of time.

Averaging Algorithm

- Both Cristian's algorithm and the Berkeley algorithm are centralized algorithms with the disadvantages such as the existence of the single point of failure and high traffic volume around the server.

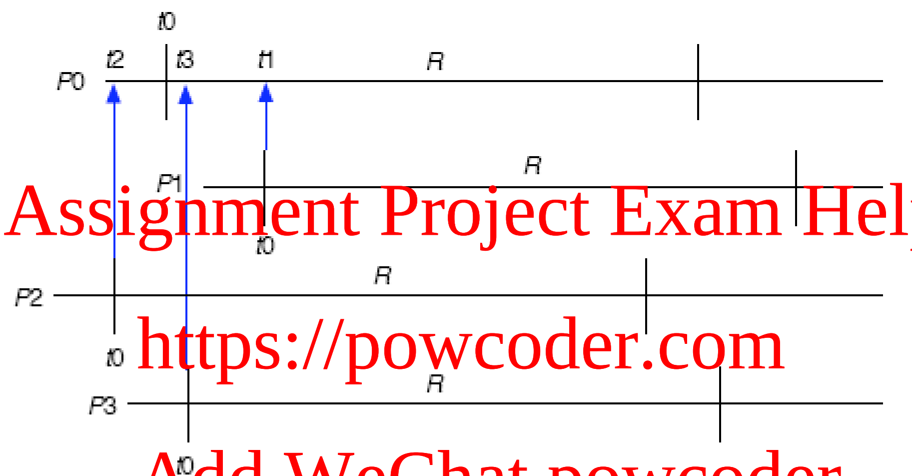
- "Averaging algorithm" is a decentralized algorithm.

- This algorithm divides time into resynchronization intervals with a fixed length R.

- Every machine broadcasts the current time at the beginning of each interval according to its clock.

- A machine collects all other broadcasts for a certain interval and sets the local clock by the average of their arrival times.

Averaging Algorithm



- Clock on processor *P*0 should be advanced by Δ*t*0 as below

$$\Delta t_0 = \frac{t_0 + t_1 + t_2 + t_3}{4} - t_0$$

Logical Clock and Physical Clock

- Lamport showed
  - Clock synchronization need not be absolute
  - If two processes do not interact their clocks need not be synchronized.
  - What matters is they agree on the order in which events occur.

- For many purposes, it is sufficient that all interacting machines agree on the same time. It is not essential that this time is the real time.
  - The clocks that agree among certain computers but not necessarily with the real clock are logical clocks.
- Clocks that agree on time and within a certain time limit, are physical clocks.

Lamport's Synchronization Algorithm

- This algorithm only determines event order, but does not synchronize clocks.

- "Happens-before" relation:
  – "$A \rightarrow B$" is read "A happens before B": This means that *all processes agree* that event A occurs before event B.

- The happens-before relation can be observed directly in two situations:
  – If A and B are events in the same process, and A occurs before B, then $A \rightarrow B$.
  – If A is the event of a message being sent by one process, and B is the event of the message being received by another process, then $A \rightarrow B$.

- "Happens-before" is a transitive relation.

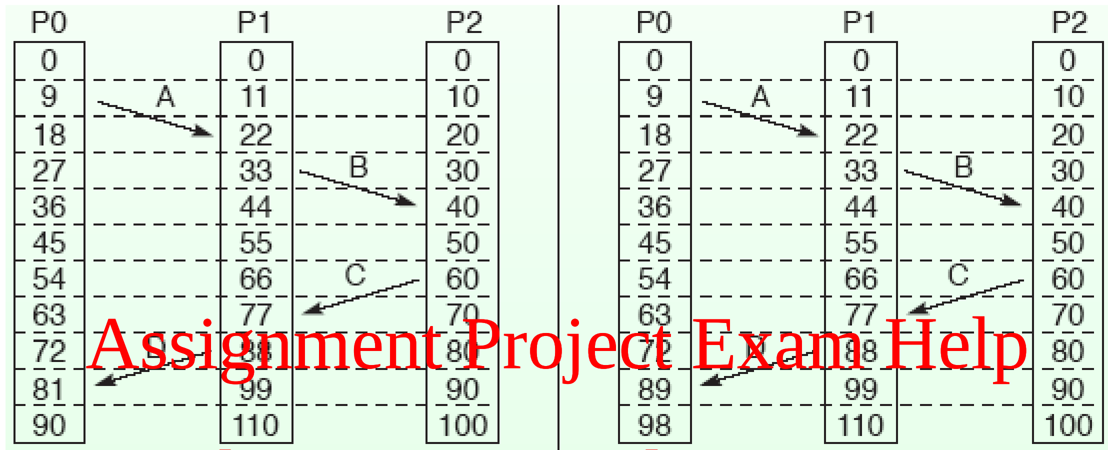
Lamport's Synchronization Algorithm

- If two events, X and Y happen in different processes that do not exchange messages (not even indirectly via third parties), then neither $X \rightarrow Y$ nor $Y \rightarrow X$ is true. These events are "concurrent."

- What we need is a way to assign a time value $C(A)$ on which all processes agree for every event A. The time value must have the following properties:
  - If $A \rightarrow B$, then $C(A) < C(B)$.
  - The clock time must always go forward, never backward.

- Suppose there are three processes which run on different machines as in the following figure. Each processor has its own local clock. The rates of the local clocks are different.

Lamport's Synchronization Algorithm



- By setting the clocks as below, we can define total ordering of all events.

- If event *A* happens before event *B* within the same process, *C*(*A*) < *C*(*B*) is satisfied.
- If event *A* and event *B* represent the sending and receiving of a message, the clock of the receiving side is set so that *C*(*A*) < *C*(*B*).
- For all events, the clock is increased at least by 1 between two events.

Shortcoming of Lamport's clock

- Lamport's clock observes if $a \rightarrow b$ then $C(a) < C(b)$ but $C(a) < C(b)$ does not imply $a \rightarrow b$ always. Hence, we cannot deduce causal dependencies from time stamps.

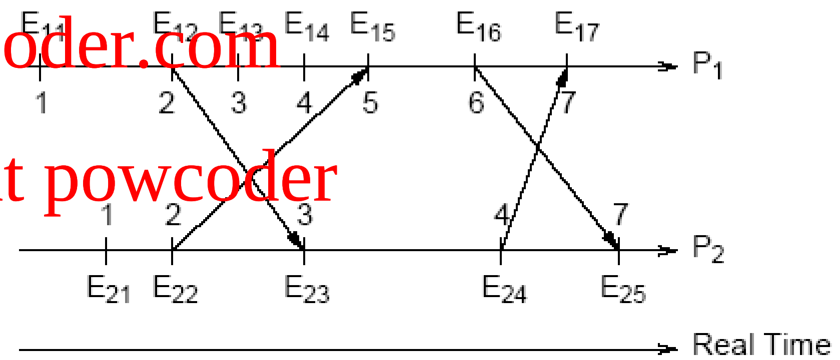
- The root of the problem is that clocks advance independently or via messages, but there is no history as to where the advance comes from.

- In this figure,

  $E_{12} \rightarrow E_{23}$ implies $C_1(E_{12}) < C_2(E_{23})$

  However we may have $E_{13} \not\rightarrow E_{24}$

  while $C_1(E_{13}) < C_2(E_{24})$



- In some situations (e.g., to implement distributed locks), a partial ordering on events is not sufficient and a total ordering is required.

Vector Clock

- Each process maintains a vector clock $\boldsymbol{V_i}$ of size $\boldsymbol{N}$, where $\boldsymbol{N}$ is the number of processes. The component $\boldsymbol{V_i}[\boldsymbol{j}]$ contains the process $\boldsymbol{p_i}$' s knowledge about $\boldsymbol{p_j}$' s clock. Initially, we have $\boldsymbol{V_i}[\boldsymbol{j}] := 0$ for $\boldsymbol{i, j} \,\epsilon\, \{1, 2, \ldots, N\}$

- Clocks are advanced as follows:

1. Before $\boldsymbol{p_i}$ timestamps an event, it executes $\boldsymbol{V_i[i]} = \boldsymbol{V_i[i]} + 1$.

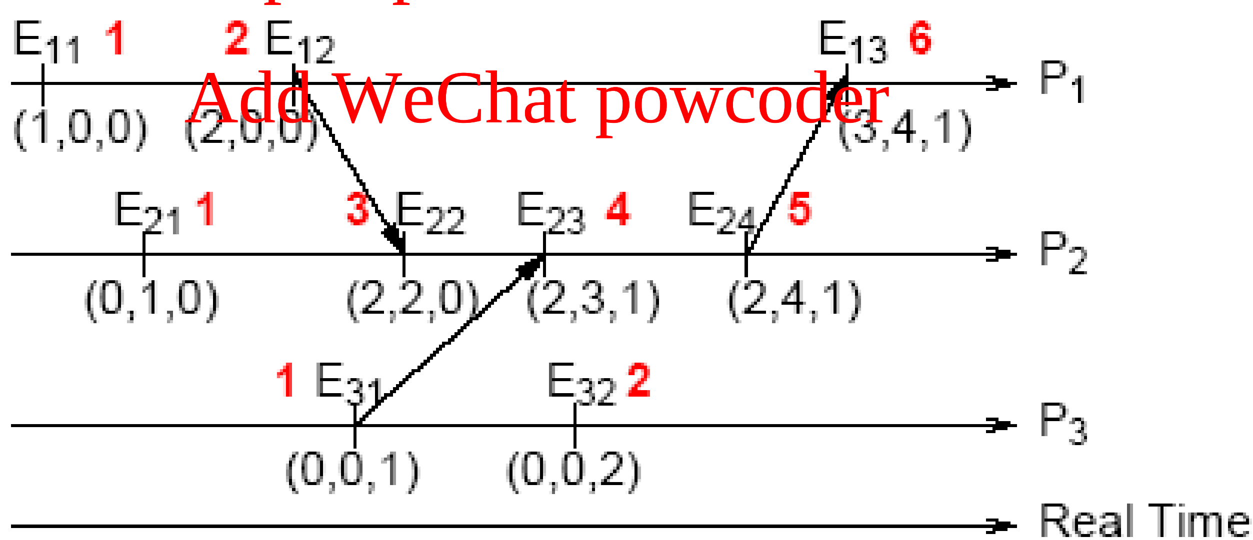
2. Whenever a message $\boldsymbol{m}$ is sent from $\boldsymbol{p_i}$ to $\boldsymbol{p_j}$:
   - Process $\boldsymbol{p_i}$ executes $\boldsymbol{V_i[i] := V_i[i] + 1}$ and sends $\boldsymbol{V_i}$ with $\boldsymbol{m}$.
   - Process $\boldsymbol{p_j}$ receives $\boldsymbol{V_i}$ with $\boldsymbol{m}$ and merges the vector clocks $\boldsymbol{V_i}$ and $\boldsymbol{V_j}$ as follows:

$$\text{Vj [k]} := \begin{cases} \text{Max}(\boldsymbol{V_j[k]}, \boldsymbol{V_i[k]})+1 & \text{, if } \boldsymbol{j}=\boldsymbol{k} \text{ (as in scalar clocks)} \\ \text{Max}(\boldsymbol{V_j[k]}, \boldsymbol{V_i[k]}) & \text{, otherwise} \end{cases}$$

This last part ensures that everything that subsequently happens at p_j is now causally related to everything that previously happened at p_i.

Vector Clock

- Each event is annotated with both its vector clock value (the triple) and the corresponding value of a scalar Lamport clock (Red color).
- For $C_1(E_{12})$ and $C_3(E_{32})$, we have 2 = 2 versus (2, 0, 0) ≠ (0, 0, 2). Likewise we have $C_2(E_{24}) > C_3(E_{32})$ but (2, 4, 1) NOT≥ (0, 0, 2) and thus E_{32} NOT$\rightarrow E_{24}$



Mutual exclusion in DS

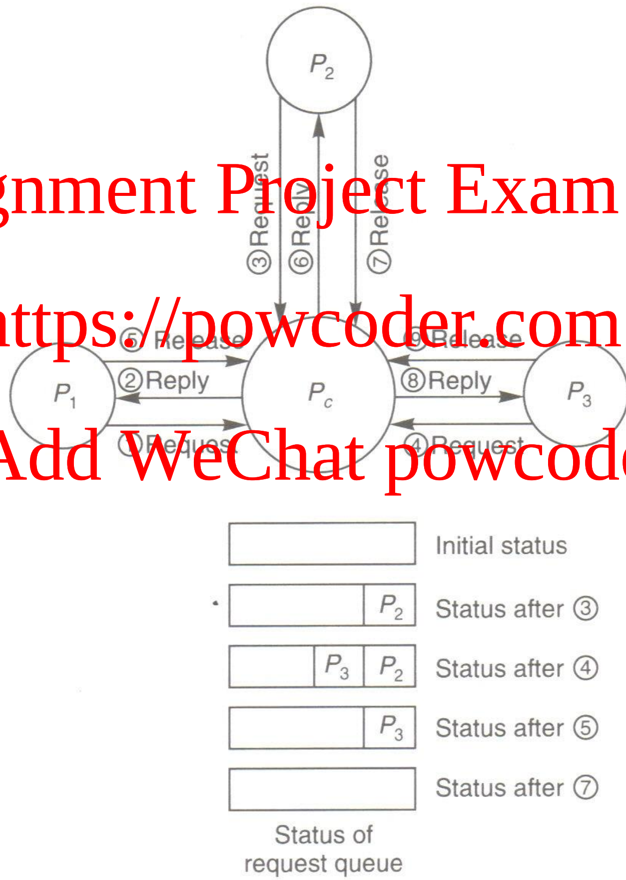
- When multiple processes access shared resources, using the concept of *critical sections* is a relatively easy way to program access of the shared resources.
  - Critical section is a section in a program that accesses shared resources.
  - A process enters a critical section before accessing the shared resource to ensure that no other process will use the shared resource at the same time.

- Critical sections are protected using semaphores and monitors in single-processor systems. We cannot use these mechanisms in distributed systems.

A Centralized Algorithm

- This algorithm simulates mutual exclusion in single processor systems.
- One process is elected as the coordinator.
- When a process wants to enter a critical section, it sends a request to the coordinator stating which critical section it wants to enter.
- If no other process is currently in that critical section, the coordinator returns a reply granting permission.
- If a different process is already in the critical section, the coordinator queues the request.
- When the process exits the critical section, the process sends a message to the coordinator releasing its exclusive access.
- The coordinator takes the first item off the queue of deferred request and sends that process a grant message.

A Centralized Algorithm



A Centralized Algorithm

Advantages

- Since the service policy is first-come first-serve, it is fair and no process waits forever.
- It is easy to implement.
- It requires only three messages, request, grant, and release, per use of a critical section.

Disadvantages

- If the coordinator crashes, the entire system may go down.
- Processes cannot distinguish a dead coordinator from “permission denied.”
- A single coordinator may become a performance bottleneck.

A Distributed Algorithm

- The distributed algorithm proposed by Ricart and Agrawala requires ordering of all events in the system. We can use the Lamport's algorithm for the ordering.

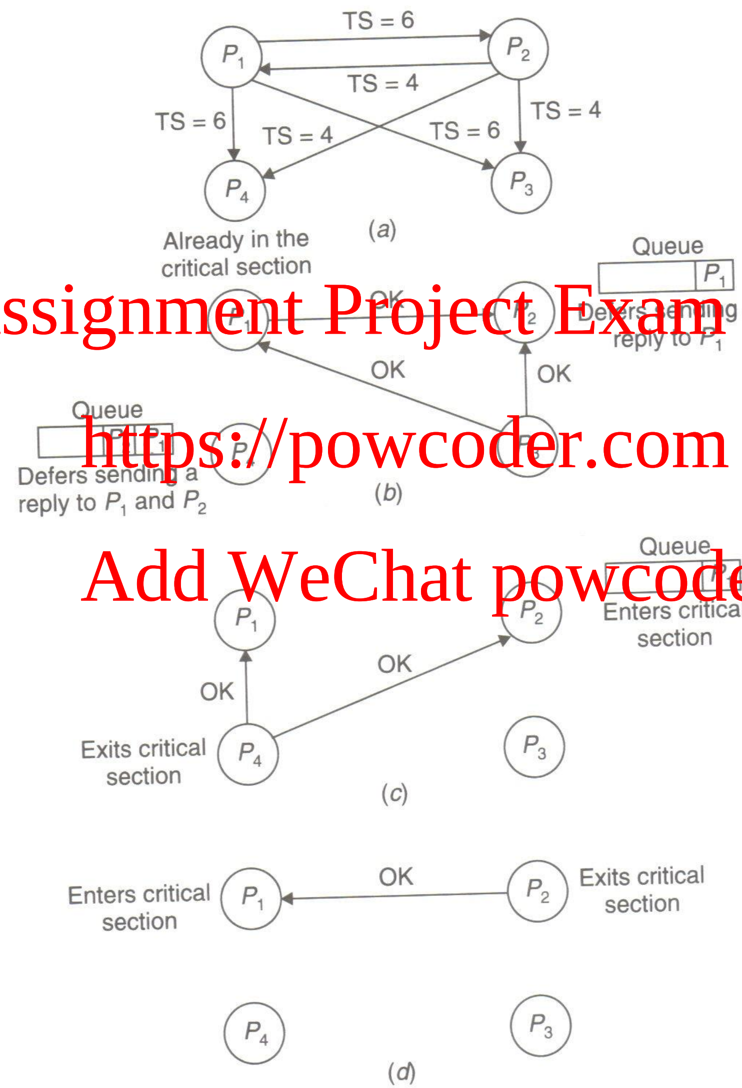
- When a process wants to enter a critical section, the process sends a request message to all other processes. The request message includes
  - Name of the critical section
  - Process number
  - Current time

- The other processes receive the request message.
  - If the process is not in the requested critical section and also has not sent a request message for the same critical section, it returns an OK message to the requesting process.
  - If the process is in the critical section, it does not return any response and puts the request to the end of a queue.
  - If the process has sent out a request message for the same critical section, it compares the time stamps of the sent request message and the received message.

A Distributed Algorithm

- If the time stamp of the received message is smaller than the one of the sent message, the process returns an OK message.
- If the time stamp of the received message is larger than the one of the sent message, the request message is put into the queue.
- The requesting process waits until all processes return OK messages.
- When the requesting process receives all OK messages, the process enters the critical section.
- When a process exits from a critical section, it returns OK messages to all requests in the queue corresponding to the critical section and removes the requests from the queue.
- Processes enter a critical section in time stamp order using this algorithm.

A Distributed Algorithm



A Distributed Algorithm

- Advantage
  - No deadlock or starvation happens.

- Disadvantages
  - 2(n − 1) messages are required to enter a critical section. Here n is the number of processes.
  - If one of the processes fails, the process does not respond to the request. It means no process can enter the critical section.
  - The coordinator is the bottleneck in a centralized system. Since all processes send requests to all other processes in this distributed algorithm, all processes are bottlenecks in this algorithm.

Token Ring Algorithm

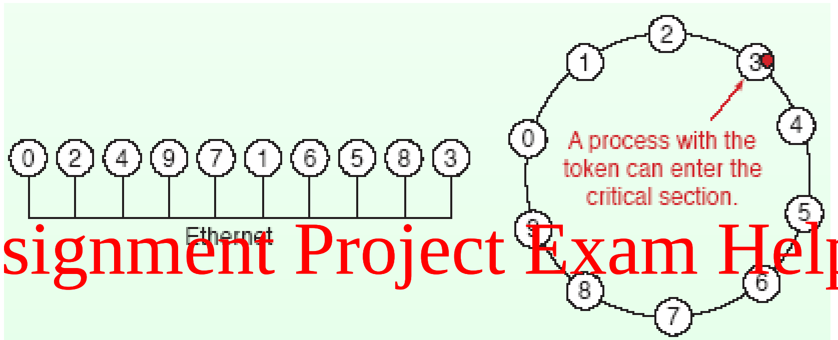
- We can construct a virtual ring by assigning a sequence number to processes.
  – Processes can form a virtual ring even if the processes are not physically connected in a ring shape. The process #0 receives a token when the ring is initialized.

- The token is passed to the process with the next sequence number.

- A process can enter the critical section only if the process holds the corresponding token. The process passes the token to the next process when it is done.

- A process passes the received token if it needs not to enter the critical section upon receiving the token.

Token Ring Algorithm



Advantage

Processes don't suffer from starvation. Before entering a critical section, a process waits at most for the duration that all other processes enter and exit the critical section.

Disadvantages

- If a token is lost for some reasons, another token must be generated.
- Detecting lost token is difficult since there is no upper bound in the time a token takes to rotate the ring.
- The ring must be reconstructed when a process crashes.

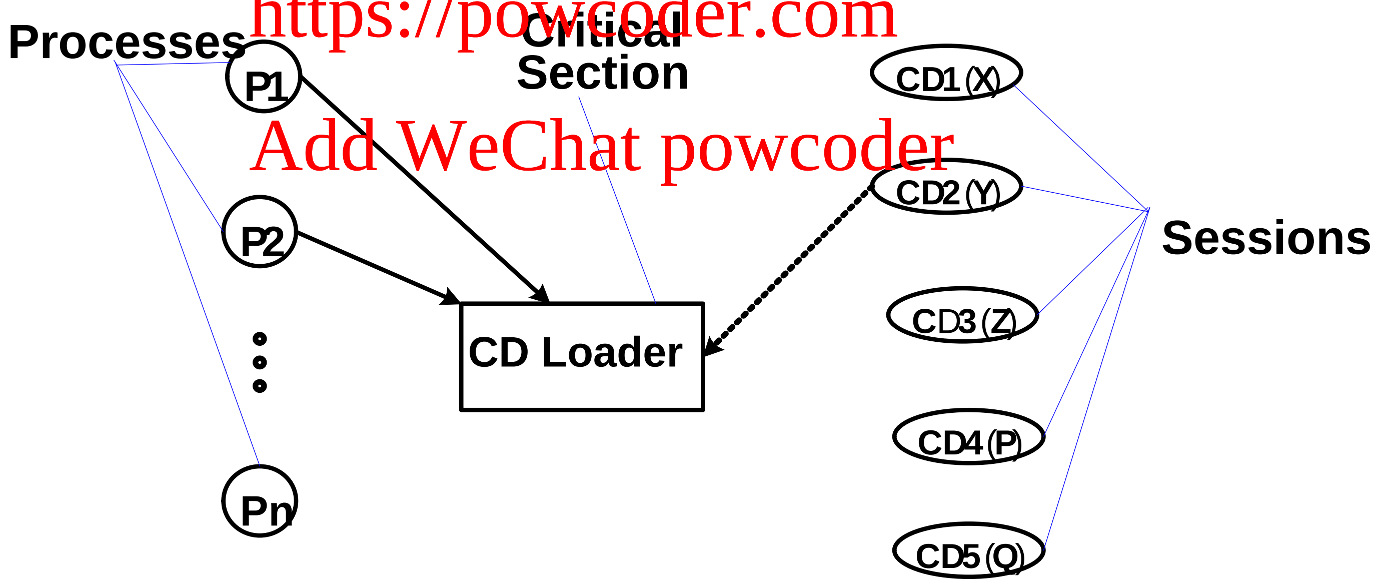
Mutual Exclusion Algorithm: A Comparison

- Messages Exchanged (Messages per entry/exit of critical section)
  - Centralized: 3
  - Distributed: 2(n − 1)
  - Ring: 2

- Reliability (Problems that may occur)
  - Centralized: coordinator crashes
  - Distributed: any process crashes
  - Ring: lost token, process crashes

Group Mutual Exclusion

- In the group mutual exclusion problem, which generalizes mutual exclusion, processes choose 'session' when they want entry to the Critical Section (CS); processes are allowed to be in the CS simultaneously provided they request the same session.



Applications of GME

- In some applications such as Computer Supported Cooperative Work (CSCW)

- Wireless application

- An efficient GME solution could also help improve the quality of services of an Internet server. The GME protocol can be used to group different requests for the same service, and thereby, reduce the memory swapping.

Deadlocks in Distributed Systems

- A *deadlock* is a condition where a process cannot proceed because it needs to obtain a resource held by another process and it itself is holding a resource that the other process needs.

- We can consider two types of deadlock:
  - **communication deadlock** occurs when process A is trying to send a message to process B, which is trying to send a message to process C which is trying to send a message to A.

  - A **resource deadlock** occurs when processes are trying to get exclusive access to devices, files, locks, servers, or other resources.

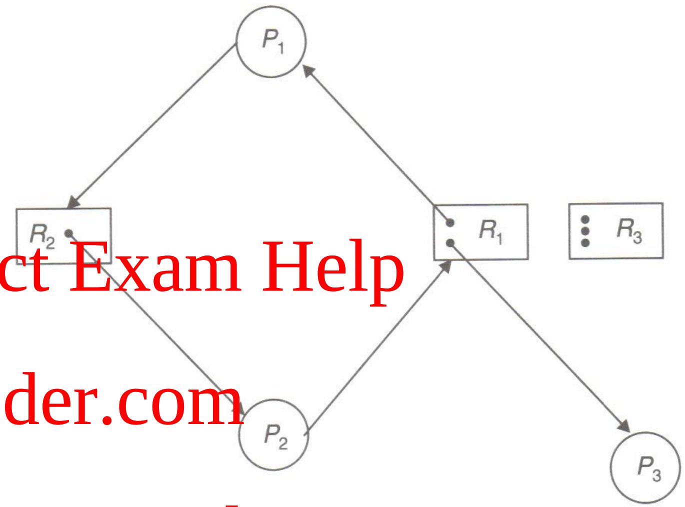
- We will not differentiate between these types since we can consider communication channels to be resources without loss of generality.

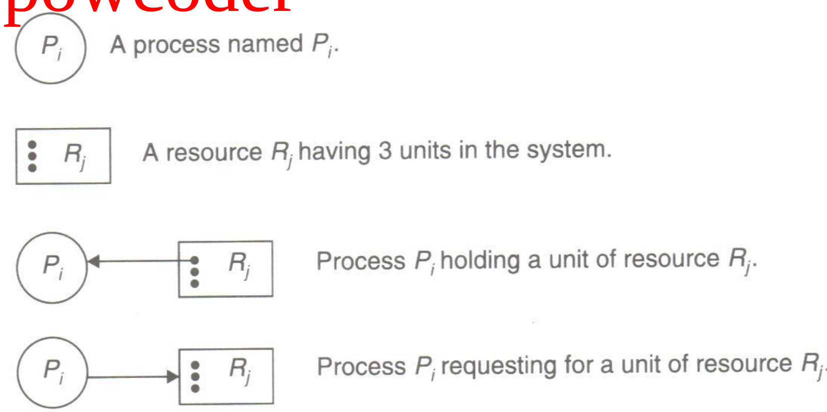
Necessary conditions for Deadlock

- Four conditions have to be met for deadlock to be present:
  - **Mutual exclusion.** A resource can be held by at most one process
  - **Hold and wait.** Processes that already hold resources can wait for another resource.
  - **Non-preemption.** A resource, once granted, cannot be taken away from a process.
  - **Circular wait.** Two or more processes are waiting for resources held by one of the other processes.

Deadlock Modeling

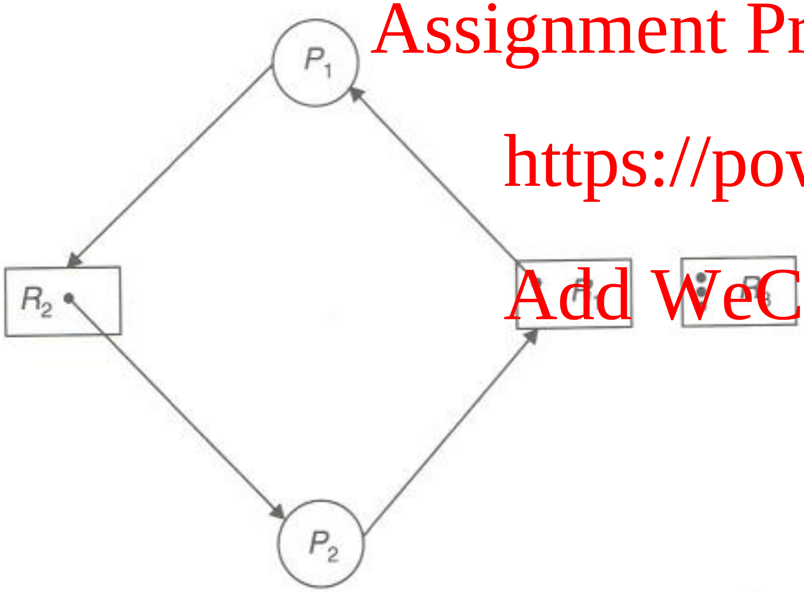
- For deadlock modeling, a directed graph called a *resource allocation graph*, is used in which both the sets of nodes and edges are partitioned into two types, resulting in following graph elements-
  - **Process nodes: P_1, P_2, P_3**
  - **Resource nodes: R_1, R_2, R_3**
  - **Assignment edges: (R_1, P_1), (R_1, P_3)**
  - **Request edges: (P_1, R_2), (P_2, R_1)**



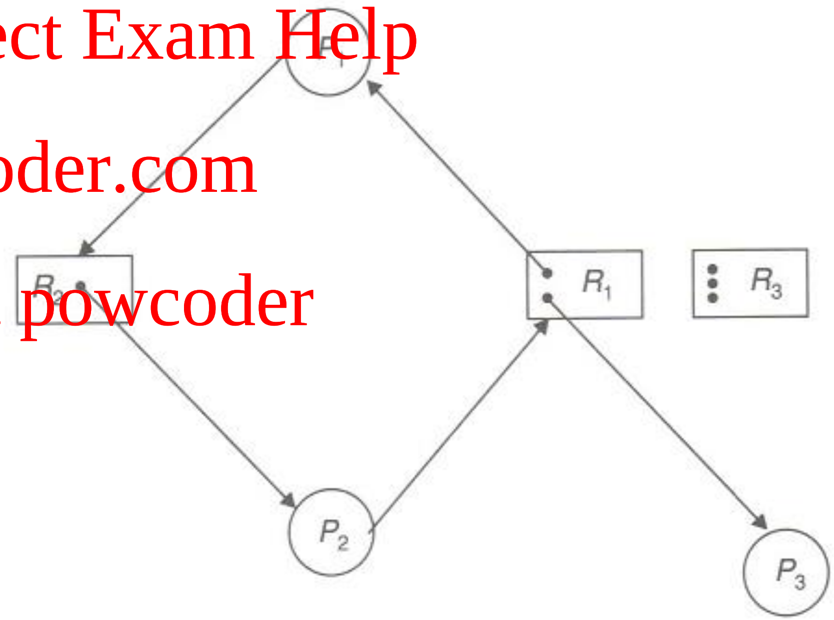


Necessary Conditions for Deadlock

- In resource allocation graph, a *cycle* is a necessary condition for a deadlock to exist.



It is a Deadlock



But this is not

Sufficient condition for a deadlock

- If there is only one copy of all resources
  – **A Cycle** in resource allocation graph

- If there are multiple copies of some resources

  – **A Knot** in resource allocation graph

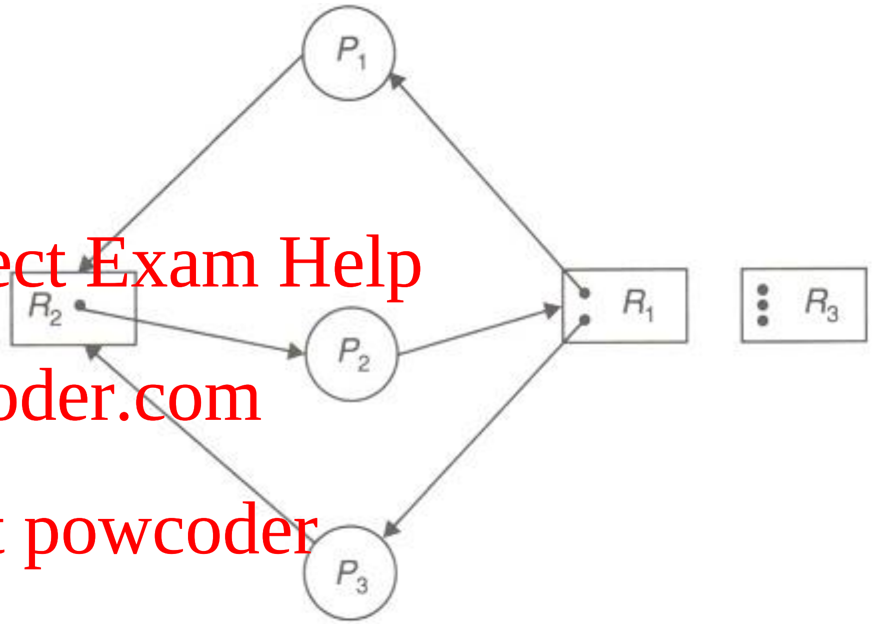
- Note-

  A **knot** in a directed graph is a collection of vertices and edges with the property that ***every vertex in the knot has outgoing edges, and all outgoing edges from vertices in the knot terminate at other vertices in the knot***.

Sufficient condition for a deadlock

Cycles:

1. (P_1, R_2, P_2, R_1, P_1)
2. (P_3, R_2, P_2, R_1, P_3)



Knot:

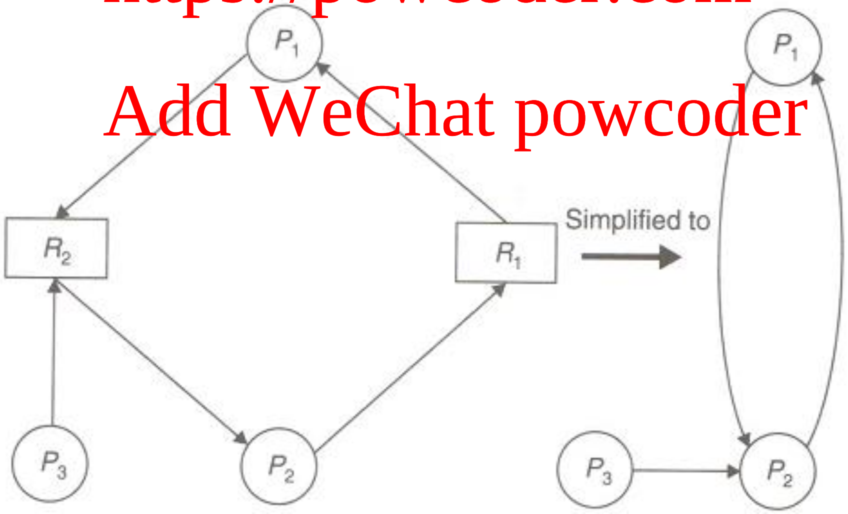
1. {P_1, P_2, P_3, R_1, R_2}

Thus here is a deadlock !

Wait for Graph

- When all resource types have only a single unit each, a simplified form of resource allocation graph is used – wait-for graph.

- Wait-for graph may be constructed from resource allocation graph by removing the resource nodes and collapsing the appropriate edges.

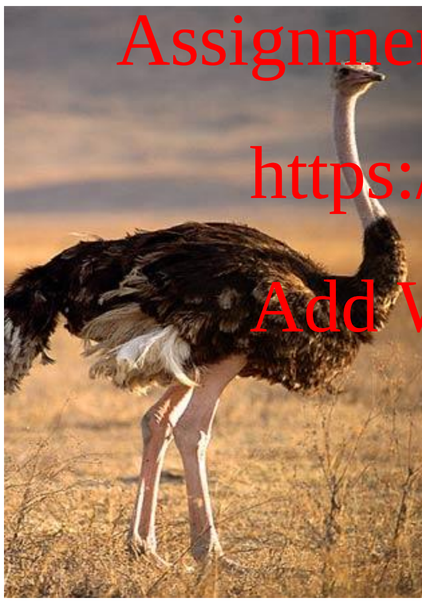


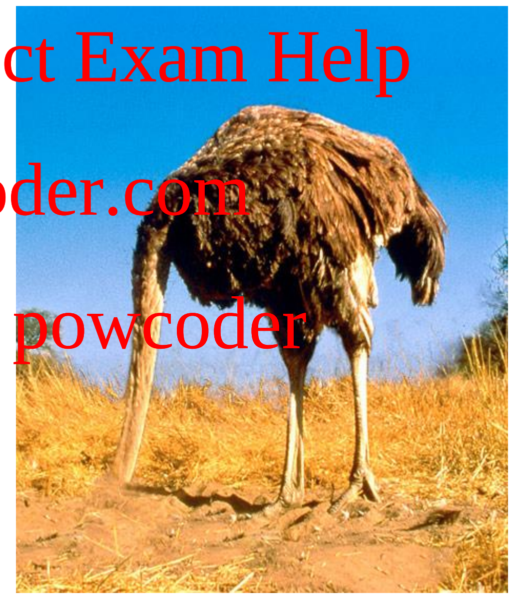
Handling deadlocks in DS

- Strategies
  – Ostrich algorithm
  – Deadlock detection and recovery
  – Deadlock prevention by careful resource allocations
  – Deadlock avoidance by designing the system in such a way that deadlocks become impossible to occur

Ostrich Algorithm

- Ignore the deadlock problem





Deadlock Detection in DS

- Preventing or avoiding deadlocks can be difficult.
- Detecting them is easier.
- When deadlock is detected
  - kill off one or more processes
    - annoyed users
  - if system is based on atomic transactions, abort one or more transactions
    - transactions have been designed to withstand being aborted
    - system restored to state before transaction began
    - transaction can start a second time
    - resource allocation in system may be different so the transaction may succeed

Centralized deadlock detection Algorithm

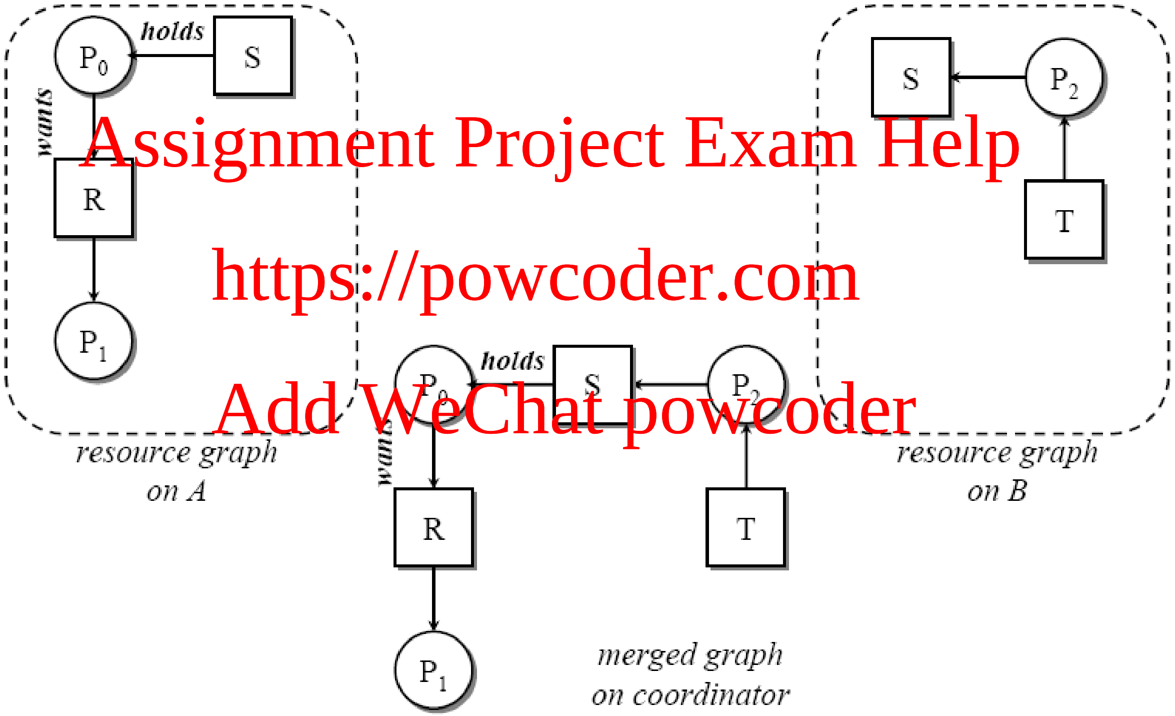
- Imitate the non-distributed algorithm through a coordinator

- Each machine maintains a resource graph for its processes and resources

- A **central coordinator** maintains a graph for the entire system

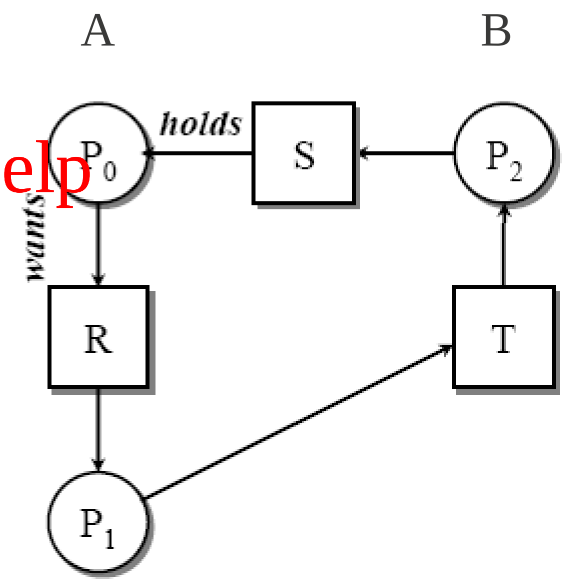
  - message can be sent to coordinator each time an arc is added or deleted
  - list of arc adds/deletes can be sent periodically

Centralized deadlock detection Algorithm



False Deadlock by Centralized Algorithm

- Two events occur:
  1. Process P1 releases resource R
  2. Process P1 asks machine B for resource T
- Two messages are sent to the coordinator:
  1 (from A): *releasing R*
  2 (from B): *waiting for T*
- If message 2 arrives first, the coordinator constructs a graph that has a cycle and hence detects a deadlock. This is **false deadlock.**



- Global time ordering must be imposed on all machines or
- Coordinator can reliably ask each machine whether it has any release messages.

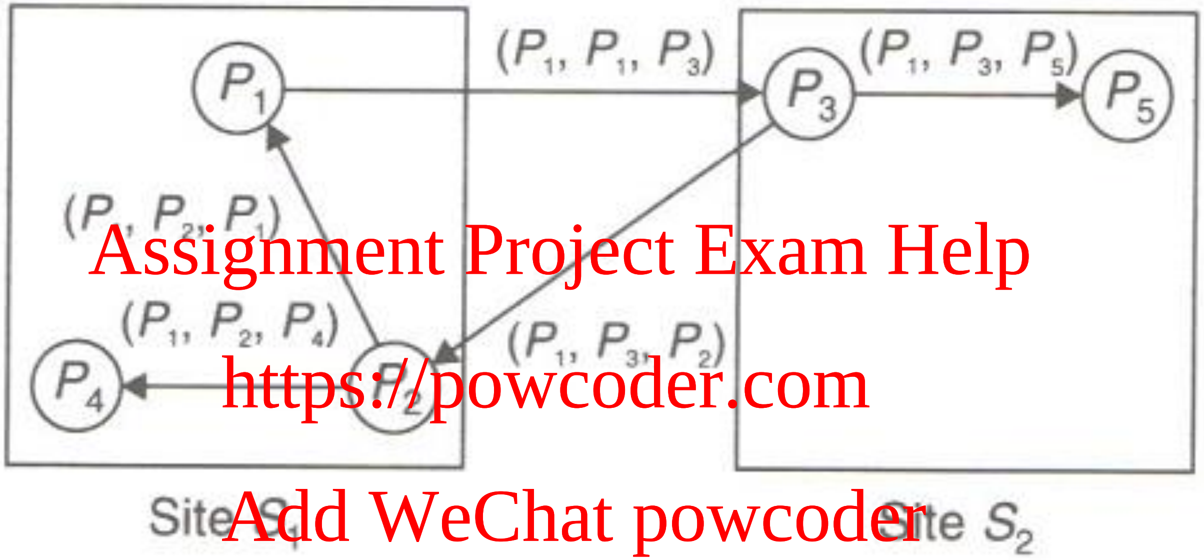
A Distributed Approach

- Proposed by Chandy, Misra and Haas.
- Processes can requests multiple resources at once
- Some processes wait for local resources
- Some processes wait for resources on other machines
- Algorithm invoked when a process has to wait for a resource
- A probe message is generated. This message contains 3 fields-
  - process that just blocked
  - process sending the message
  - process to whom it is being sent

Chandy-Misra-Haas Algorithm

- when ***probe*** message arrives, recipient checks to see if it is waiting for any processes
  - If no, ignore the message
  - If so, update message
    - replace second field by its own process number
    - replace third field by the number of the process it is waiting for
    - send messages to each process on which it is blocked
- If a message goes all the way around and comes back to the original sender, a cycle exists
- That means, *we have deadlock*

CMH Algorithm



- Suppose,
  - P_1 gets blocked by P_3
  - P_3 gets blocked by P_2 and P_5
  - P_2 gets blocked by P_1 and P_4

Recovery after Detection

- One of the following methods may be used
  - Asking for operator intervention
  - Terminating of process(es)
  - Rollback of process(es)

- Issues in recovery from deadlock
  - Minimization of recovery cost
  - Prevention of starvation

Deadlock Avoidance

- When a process requests a resource, even if the resource is available, it is not immediately allocated to the process. Rather, the system simply assumes that a request is granted

- With the assumption made is previous step and advance knowledge of resource usage of processes, the system performs some analysis to decide whether granting the request is safe or unsafe.

- The resource is allocated to the process only when the analysis of previous step shows that it I safe to do so, otherwise the request is deferred.

- Although theoretically attractive, these algorithms are rarely used.

Deadlock prevention

Collective requests

- Denies the hold-and-wait condition. Two ways-
  - A process must request all of its resources before it begins the execution. If all resources are granted, execution starts; otherwise the process would just wait.
  - Instead of requesting all its resources before, a process may request a resource during its execution if it obeys that it holds no other resources while requesting. If it is holding some resources, it may release all the resources first and then request for all.

Deadlock prevention

Ordered requests

- Denies circular wait
  - All resources are numbered. If a process is holding a resource i it may request a resource having the number j only if $j>i$.

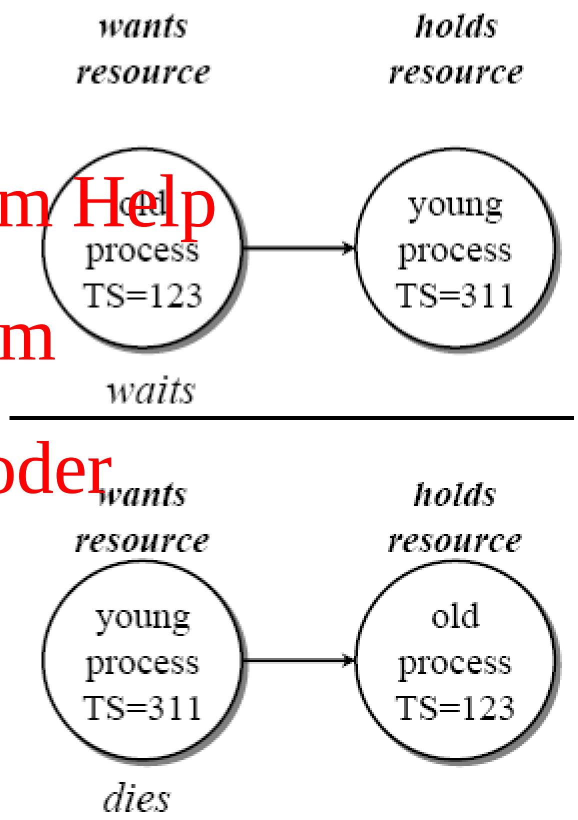
Preemption

- Resources may by preempted away from one process and assigned to another.
- Two widely used methods are *wait-die* and *wound-wait*

Wait-die Algorithm

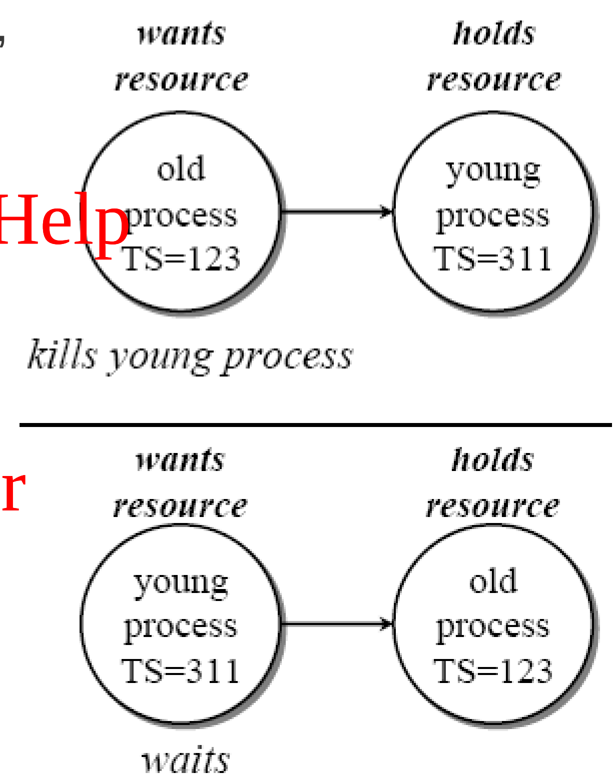
- Old process wants resource held by a younger process
  - – old process waits

- Young process wants resource held by older process
  - – young process kills itself



Wound-wait Algorithm

- Instead of killing the transaction making the request, kill the resource owner
- Old process wants resource held by a younger process
  - – old process kills the younger process
- Young process wants resource held by older process
  - – young process waits



Summary

- Three Real Clock Synchronisation Methods?
  - Cristian’s Method
  - Berkeley Algorithm
  - Averaging Algorithm
- Two Logical (Clock) Synchronisation Techniques
  - Lamport
  - Vector Clock
- Three Mutual Exclusion Approaches?
  - Centralised
  - Distributed
  - GME
- Main Deadlock Modeling Areas?
  - Necessary condition for occurrence
  - Deadlock Detection
  - Deadlock Handling

References

- I Kuz, M Chakravarty, G Heiser, Lecture notes, University of NSW, 2010.
- K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. ACM Transactions on Computer Systems, 3:63–75, 1985.
- Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. Com- munications of the ACM, 21:558–565, 1978.
- G. Ricart and A. Argawala. An optimal algorithm for mutual exclusion in computer networks. Communications of the ACM, 24(1), January 1981.