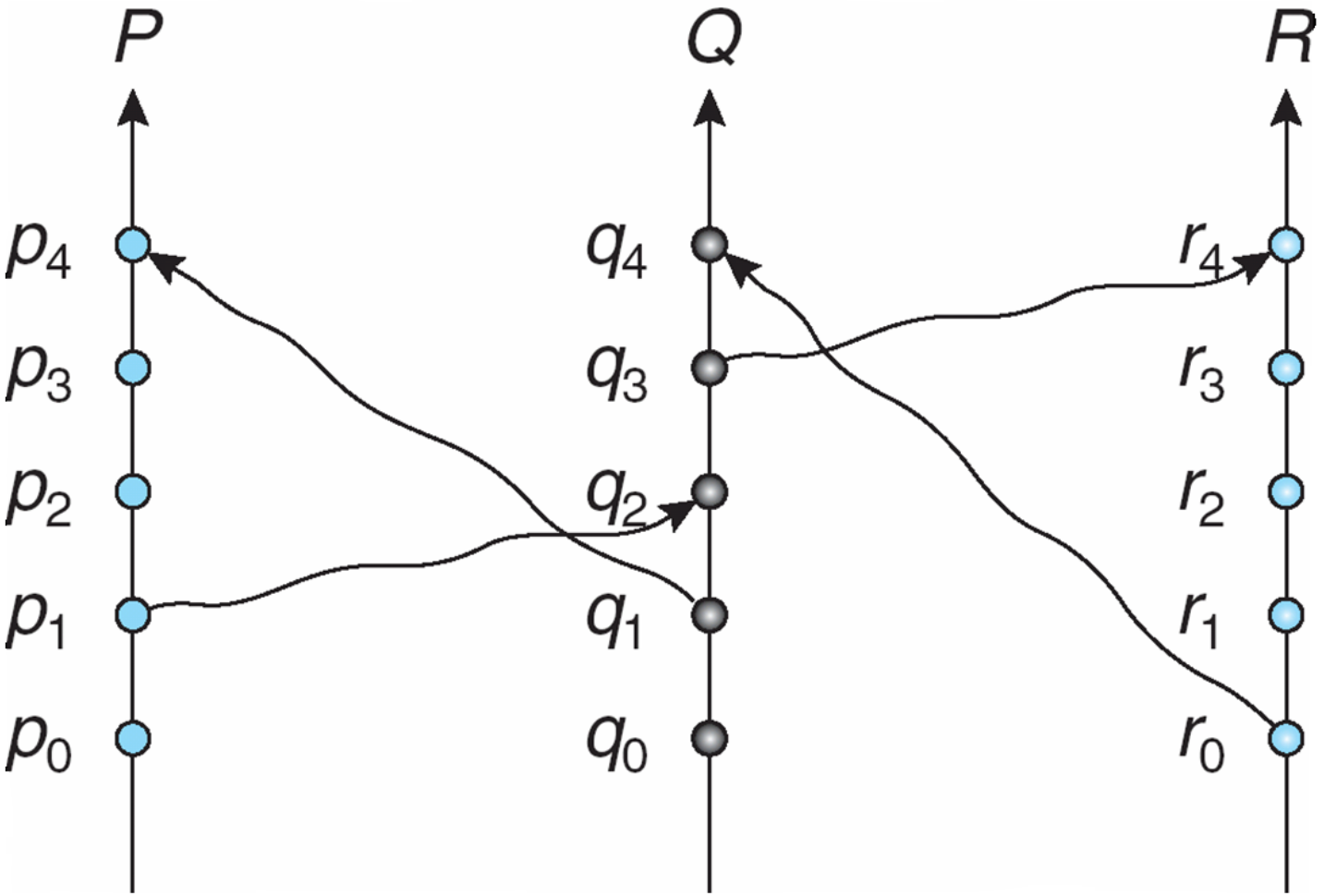
Event Ordering

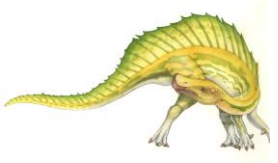
- Happened-before relation (denoted by $\rightarrow$)
  - If A and B are events in the same process, and A was executed before B, then $A \rightarrow B$
  - If A is the event of sending a message by one process and B is the event of receiving that message by another process, then $A \rightarrow B$
  - If $A \rightarrow B$ and B $\rightarrow C$ then $A \rightarrow C$

Relative Time for Three Concurrent Processes



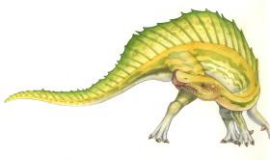
Implementation of $\rightarrow$

- Associate a timestamp with each system event
  - Require that for every pair of events A and B, if A $\rightarrow$ B, then the timestamp of A is less than the timestamp of B
- Within each process Pi a logical clock, LCi is associated
  - The logical clock can be implemented as a simple counter that is incremented between any two successive events executed within a process
    - Logical clock is monotonically increasing
- A process advances its logical clock when it receives a message whose timestamp is greater than the current value of its logical clock
- If the timestamps of two events A and B are the same, then the events are concurrent
  - We may use the process identity numbers to break ties and to create a total ordering



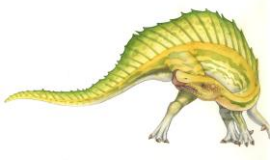
Distributed Mutual Exclusion (DME)

- Assumptions
  - The system consists of n processes; each process P_i resides at a different processor
  - Each process has a critical section that requires mutual exclusion
- Requirement
  - If P_i is executing in its critical section, then no other process P_j is executing in its critical section



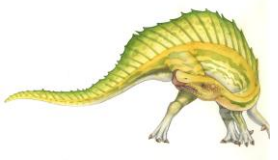
DME: Centralized Approach

- One of the processes in the system is chosen to coordinate the entry to the critical section
- A process that wants to enter its critical section sends a request message to the coordinator
- The coordinator decides which process can enter the critical section next, and its sends that process a reply message
- When the process receives a reply message from the coordinator, it enters its critical section
- After exiting its critical section, the process sends a release message to the coordinator and proceeds with its execution
- This scheme requires three messages per critical-section entry:
  - request
  - reply
  - release



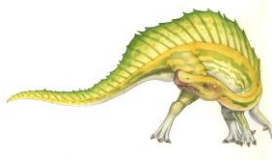
DME: Fully Distributed Approach

- When process P_i wants to enter its critical section, it generates a new timestamp, *TS*, and sends the message *request* (P_i, *TS*) to all other processes in the system
- When process P_j receives a *request* message, it may reply immediately or it may defer sending a reply back
- When process P_i receives a *reply* message from all other processes in the system, it can enter its critical section
- After exiting its critical section, the process sends *reply* messages to all its deferred requests



DME: Fully Distributed Approach (Cont)

- The decision whether process P_j replies immediately to a *request*(P_i, TS) message or defers its reply is based on three factors:
  - If P_j is in its critical section, then it defers its reply to P_i
  - If P_j does *not* want to enter its critical section, then it sends a *reply* immediately to P_i
  - If P_j wants to enter its critical section but has not yet entered it, then it compares its own request timestamp with the timestamp TS
    - If its own request timestamp is greater than TS, then it sends a *reply* immediately to P_i (P_i asked first)
    - Otherwise, the reply is deferred

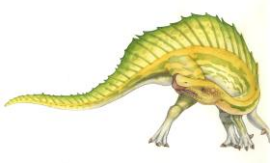


Desirable Behavior of Fully Distributed Approach

- Freedom from Deadlock is ensured
- Freedom from starvation is ensured, since entry to the critical section is scheduled according to the timestamp ordering
  - The timestamp ordering ensures that processes are served in a first-come, first served order
- The number of messages per critical-section entry is

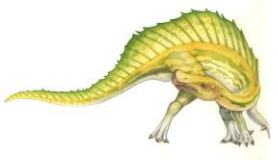
$$2 \times (n - 1)$$

  This is the minimum number of required messages per critical-section entry when processes act independently and concurrently



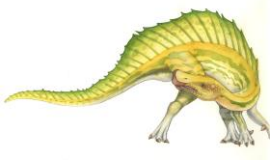
Three Undesirable Consequences

- The processes need to know the identity of all other processes in the system, which makes the dynamic addition and removal of processes more complex
- If one of the processes fails, then the entire scheme collapses
  - This can be dealt with by continuously monitoring the state of all the processes in the system
- Processes that have not entered their critical section must pause frequently to assure other processes that they intend to enter the critical section
  - This protocol is therefore suited for small, stable sets of cooperating processes



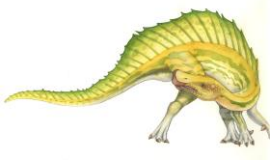
Token-Passing Approach

- Circulate a token among processes in system
  - Token is special type of message
  - Possession of token entitles holder to enter critical section
- Processes *logically* organized in a ring structure
- Unidirectional ring guarantees freedom from starvation
- Two types of failures
  - Lost token – election must be called
  - Failed processes – new logical ring established



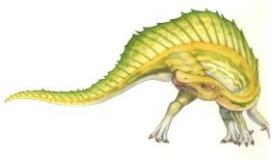
Election Algorithms

- Determine where a new copy of the coordinator should be restarted
- Assume that a unique priority number is associated with each active process in the system, and assume that the priority number of process P_i is i
- The coordinator is always the process with the largest priority number. When a coordinator fails, the algorithm must elect that active process with the largest priority number
- Two algorithms, the bully algorithm and a ring algorithm, can be used to elect a new coordinator in case of failures



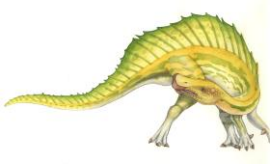
Bully Algorithm

- Applicable to systems where every process can send a message to every other process in the system
- If process P_i sends a request that is not answered by the coordinator within a time interval T, assume that the coordinator has failed; P_i tries to elect itself as the new coordinator
- P_i sends an election message to every process with a higher priority number, P_i then waits for any of these processes to answer within T



Bully Algorithm (Cont)

- If no response within T, assume that all processes with numbers greater than i have failed; P_i elects itself the new coordinator
- If answer is received, P_i begins time interval T', waiting to receive a message that a process with a higher priority number has been elected
- If no message is sent within T', assume the process with a higher number has failed; P_i should restart the algorithm

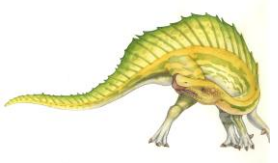


Bully Algorithm (Cont)

- If P_i is not the coordinator, then, at any time during execution, P_i may receive one of the following two messages from process P_j
  - P_j is the new coordinator ($j > i$). P_i, in turn, records this information
  - P_j started an election ($j > i$). P_i, sends a response to P_j and begins its own election algorithm, provided that *Pi* has not already initiated such an election
- After a failed process recovers, it immediately begins execution of the same algorithm
- If there are no active processes with higher numbers, the recovered process forces all processes with lower number to let it become the coordinator process, even if there is a currently active coordinator with a lower number

Ring Algorithm

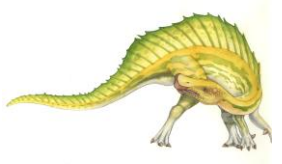
- Applicable to systems organized as a ring (logically or physically)
- Assumes that the links are unidirectional, and that processes send their messages to their right neighbors
- Each process maintains an active list, consisting of all the priority numbers of all active processes in the system when the algorithm ends
- If process Pi detects a coordinator failure, I creates a new active list that is initially empty. It then sends a message elect(i) to its right neighbor, and adds the number i to its active list



Ring Algorithm (Cont)

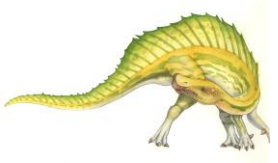
- If P_i receives a message elect(*j*) from the process on the left, it must respond in one of three ways:

  1. If this is the first *elect* message it has seen or sent, P_i creates a new active list with the numbers *i* and *j*
     - It then sends the message *elect(i),* followed by the message *elect(j)*
  2. If $i \neq j$, then the active list for P_i now contains the numbers of all the active processes in the system
     - P_i can now determine the largest number in the active list to identify the new coordinator process
  3. If $i = j$, then P_i receives the message *elect(i)*
     - The active list for P_i contains all the active processes in the system
       - P_i can now determine the new coordinator process.



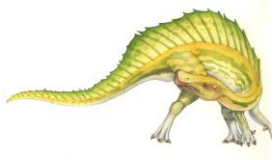
Reaching Agreement

- There are applications where a set of processes wish to agree on a common “value”
- Such agreement may not take place due to:
  - Faulty communication medium
  - Faulty processes
    - Processes may send garbled or incorrect messages to other processes
    - A subset of the processes may collaborate with each other in an attempt to defeat the scheme



Faulty Communications

- Process P_i at site A, has sent a message to process P_j at site B; to proceed, P_i needs to know if P_j has received the message
- Detect failures using a time-out scheme
  - When P_i sends out a message, it also specifies a time interval during which it is willing to wait for an acknowledgment message form P_j
  - When P_j receives the message, it immediately sends an acknowledgment to P_i
  - If P_i receives the acknowledgment message within the specified time interval, it concludes that P_j has received its message
    - If a time-out occurs, P_j needs to retransmit its message and wait for an acknowledgment
  - Continue until P_i either receives an acknowledgment, or is notified by the system that B is down

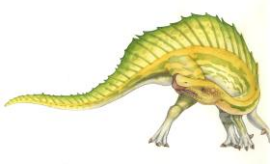


Faulty Communications (Cont)

- Suppose that P_j also needs to know that P_i has received its acknowledgment message, in order to decide on how to proceed
  - In the presence of failure, it is not possible to accomplish this task
  - It is not possible in a distributed environment for processes P_i and P_j to agree completely on their respective states

Faulty Processes (Byzantine Generals Problem)

- Communication medium is reliable, but processes can fail in unpredictable ways
- Consider a system of n processes, of which no more than m are faulty
  - Suppose that each process P_i has some private value of V_i
- Devise an algorithm that allows each nonfaulty P_i to construct a vector $X_i = (A_{i,1}, A_{i,2}, \ldots, A_{i,n})$ such that::
  - If P_j is a nonfaulty process, then $A_{ij} = V_j$.
  - If P_i and P_j are both nonfaulty processes, then $X_i = X_j$.
- Solutions share the following properties
  - A correct algorithm can be devised only if $n \geq 3 \times m + 1$
  - The worst-case delay for reaching agreement is proportionate to $m + 1$ message-passing delays



Faulty Processes (Cont)

- An algorithm for the case where m = 1 and n = 4 requires two rounds of information exchange:
  - Each process sends its private value to the other 3 processes
  - Each process sends the information it has obtained in the first round to all other processes
- If a faulty process refuses to send messages, a nonfaulty process can choose an arbitrary value and pretend that that value was sent by that process
- After the two rounds are completed, a nonfaulty process Pi can construct its vector Xi = (Ai,1, Ai,2, Ai,3, Ai,4) as follows:
  - $A_{i,j} = V_i$
  - For $j \neq i$, if at least two of the three values reported for process P_j agree, then the majority value is used to set the value of A_{ij}
    - Otherwise, a default value (*nil*) is used

End of Chapter 17