Unit: 3 Synchronization

What is Synchronization? Explain Clock Synchronization and implementation of computer clock.

- Synchronization is coordination with respect to time, and refers to the ordering of events and execution of instructions in time.
- It is often important to know when events occurred and in what order they occurred.
- Clock synchronization deals with understanding the temporal ordering of events produced by concurrent processes.
- It is useful for synchronizing senders and receivers of messages, controlling joint activity, and the serializing concurrent access to shared objects.
- The goal is that multiple unrelated processes running on different machines should be in agreement with and be able to make consistent decisions about the ordering of events in a system.
- Another aspect of clock synchronization deals with synchronizing time-of-day clocks among groups of machines.
- In this case, we want to ensure that all machines can report the same time, regardless of how imprecise their clocks may be or what the network latencies are between the machines.
- A computer clock usually consists of three components-a quartz crystal that oscillates at a well- defined frequency, a counter register, and a constant register.
- The constant register is used to store a constant value that is decided based on the frequency of oscillation of the quartz crystal.
- The counter register is used to keep track of the oscillations of the quartz crystal.
- That is, the value in the counter register is decremented by 1 for each oscillation of the quartz crystal.
- When the value of the counter register becomes zero, an interrupt is generated and its value is reinitialized to the value in the constant register.
- Each interrupt is called a clock tick.
- To make the computer clock function as an ordinary clock used by us in our day-today life, the following things are done:
    - The value in the constant register is chosen so that 60 clock ticks occur in a second.
    - The computer clock is synchronized with real time (external clock). For this, two more values are stored in the system-a fixed starting date and time and the number of ticks.
    - For example, in UNIX, time begins at 0000 on January 1, 1970.
    - At the time of initial booting, the system asks the operator to enter the current date and time.
    - The system converts the entered value to the number of ticks after the fixed starting date and time.
    - At every clock tick, the interrupt service routine increments the value of the number of ticks to keep the clock running.

Explain Drifting of Clock.

- A clock always runs at a constant rate because its quartz crystal oscillates at a well-defined frequency.
- However, due to differences in the crystals, the rates at which two clocks run are normally different from each other.
- The difference in the oscillation period between two clocks might be extremely small, but the difference accumulated over many oscillations leads to an observable difference in the times of the two clocks, no matter how accurately they were initialized to the same value.
- Therefore, with the passage of time, a computer clock drifts from the real-time clock that was used for its initial setting.
- For clocks based on a quartz crystal, the drift rate is approximately 10-6, giving a difference of 1 second

every 1,000,000 seconds, or 11.6 days.

- Hence a computer clock must be periodically resynchronized with the real-time clock to keep it non faulty.
- Even non faulty clocks do not always maintain perfect time.
- A clock is considered non faulty if there is a bound on the amount of drift from real time for any given finite time interval.
- As shown in Figure, after synchronization with a perfect clock, slow and fast clocks drift in opposite directions from the perfect clock.
- This is because for slow clocks dC/dt < 1 and for fast clocks dC/dt > 1.

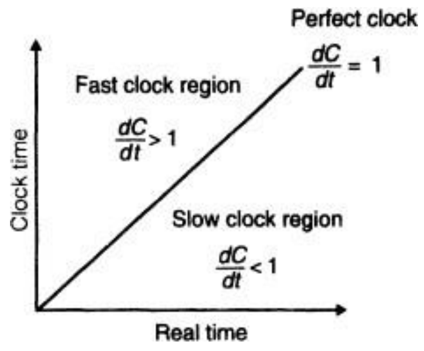


Figure: Slow, perfect, and fast clocks

- A distributed system requires the following types of clock synchronization:
  (1) Synchronization of the computer clocks with real-time (or external) clocks.
  (2) This type of synchronization is mainly required for real-time applications.
  (3) That is, external clock synchronization allows the system to exchange information about the timing of events with other systems and users.
  (4) An external time source that is often used as a reference for synchronizing computer clocks with real time is the Coordinated Universal Time (UTC).
  (5) Mutual (or internal) synchronization of the clocks of different nodes of the system.
  (6) This type of synchronization is mainly required for those applications that require a consistent view of time across all nodes of a distributed system as well as for the measurement of the duration of distributed activities that terminate on a node different from the one on which they start.

Classification of clock synchronization algorithms

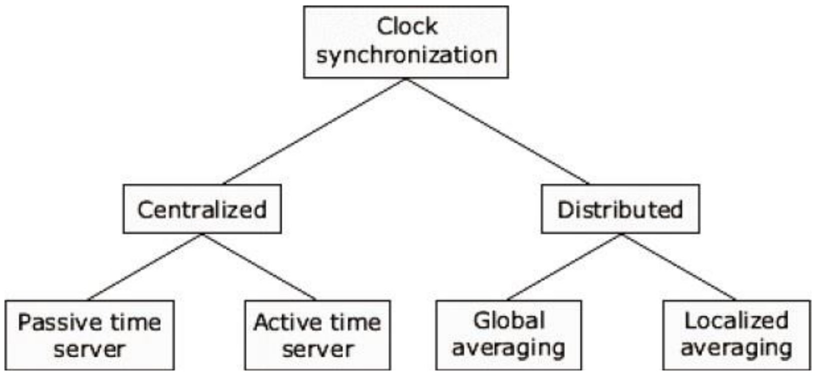
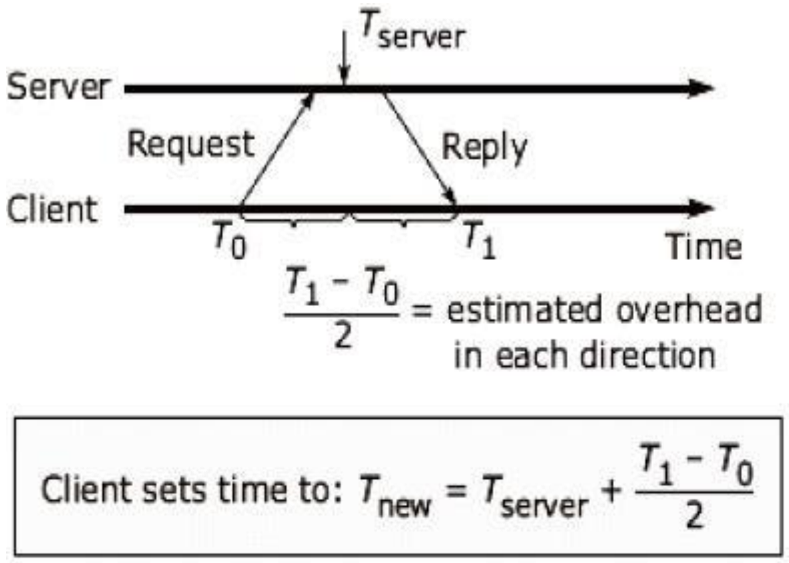


Figure: Classification of Synchronization Algorithms

Centralized Clock Synchronization Algorithms

Passive time server algorithms –

- Each node periodically sends a message called 'time=?' to the time server.
- When the time server receives the message, it responds with 'time=T' message.
- Assume that client node has a clock time of T0 when it sends 'time=?' and time T1 when it receives the 'time=T' message.
- T0 and T1 are measured using the same clock, thus the time needed in propagation of message from time server to client node would be (T1- T0)/2
- When client node receives the reply from the time server, client node is readjusted to Tserver+(T1-T0)/2.
- Two methods have been proposed to improve estimated value
    - Let the approximate time taken by the time server to handle the interrupt and process the message request message 'time=?' is equal to I.
    - Hence, a better estimate for time taken for propagation of response message from time server node to client node is taken as (T1- T0-I)/2
    - Clock is adjusted to the value Tserver+(T1- T0-I)/2



T_{server}
Server
Request
Reply
Client
T_0
T_1
Time
$\frac{T_1 - T_0}{2}$ = estimated overhead in each direction
Client sets time to: $T_{new} = T_{server} + \frac{T_1 - T_0}{2}$

Figure: Time approximation using passive time server algorithm

- **Christian method**.
    - This method assumes that a certain machine, the time server is synchronized to the UTC in some fashion called T.
    - Periodically all clock is synchronized with time server.
    - Other machines send a message to the time server, which responds with T in a response, as fast as possible.
    - The interval (T1- T0) is measured many times.
    - Those measurements in which (T1- T0) exceeds a specific threshold values are considered to be unreliable and are discarded.
    - Only those values that fall in the range (T1- T0-2Tmin) are considered for calculating the correct time.
    - For all the remaining measurements, an average is calculated which is added to T.
    - Alternatively, measurement for which value of (T1- T0) is minimum, is considered most accurate and half of its value is added to T.

Active time server algorithms

- It is also called Berkeley Algorithm.
- An algorithm for internal synchronization of a group of computers.

- A master polls to collect clock values from the others (slaves).
- The master uses round trip times to estimate the slaves' clock values.
- It obtains average from participating computers.
- It sends the required adjustment to the slaves.
- If master fails, can elect a new master to take over.
- It synchronizes all clocks to average.

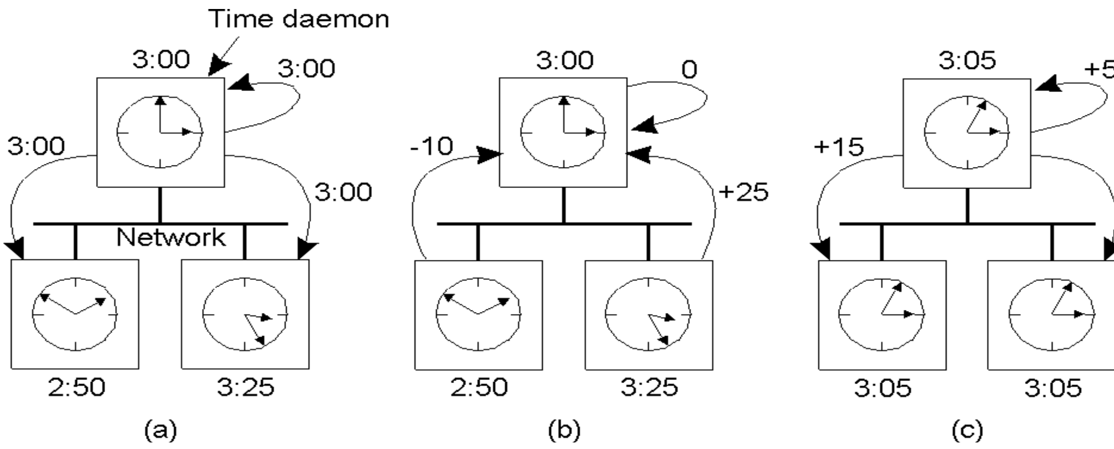


Figure: (a) The time daemon asks all the other machines for their clock values. (b) The machines answer. (c) The time daemon tells everyone how to adjust their clock.

- Here the time server (actually, a time daemon) is active, polling every machine periodically to ask what time it is there.
- Based on the answers, it computes an average time and tells all the other machines to advance their clocks to the new time or slow their clocks down until some specified reduction has been achieved.
- In Figure (a), at 3:00, the time daemon tells the other machines its time and asks for theirs.
- In Figure (b), they respond with how far ahead or behind the time daemon they are.
- Armed with these numbers, the time daemon computes the average and tells each machine how to adjust its clock Figure (c).

Distributed Clock Synchronization Algorithms – Global averaging algorithm and Local averaging algorithm

Global averaging algorithm

- One class of decentralized clock synchronization algorithms works by dividing time into fixed-length resynchronization intervals.
- The ith interval starts at T0+iR and runs until T0+(i+1) R, where T0 is an agreed upon moment in the past, and R is a system parameter.
- At the beginning of each interval, every machine broadcasts the current time according to its clock.
- Because the clocks on different machines do not run at exactly the same speed, these broadcasts will not happen precisely simultaneously.
- After a machine broadcasts its time, it starts a local timer to collect all other broadcasts that arrive during some interval 5.
- When all the broadcasts arrive, an algorithm is run to compute a new time from them.
- The simplest algorithm is just to average the values from all the other machines.
- A slight variation on this theme is first to discard the m highest and m lowest values, and average the rest.

- Discarding the extreme values can be regarded as self-defense against up to m faulty clocks sending out nonsense.
- Another variation is to try to correct each message by adding to it an estimate of the propagation time from the source.
- This estimate can be made from the known topology of the network, or by timing how long it takes for probe messages to be enclosed.

Local averaging algorithm

- This algorithm attempt to overcome the drawbacks of the global averaging algorithm.
- Nodes of a distributed system are logically arranged in some kind of pattern such as Ring.
- Periodically each node exchanges its clock time with his neighbors in the ring.
- Then its sets clock time by taking Average.
- Average is calculated by, its own clock time and clock times of its neighbors.

Mutual Exclusion and Types of Mutual Exclusion Algorithms –

- Mutual Exclusion is a process that prevents multiple threads or processes from accessing shared resources at the same time.
- 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.
- A critical section is a section in a program that accesses shared resources.
- In a distributed system, shared variables or a local kernel cannot be used to implement mutual exclusion.
- Message passing is the sole means for implementing distributed mutual exclusion.

Centralized Mutual Exclusion algorithm.

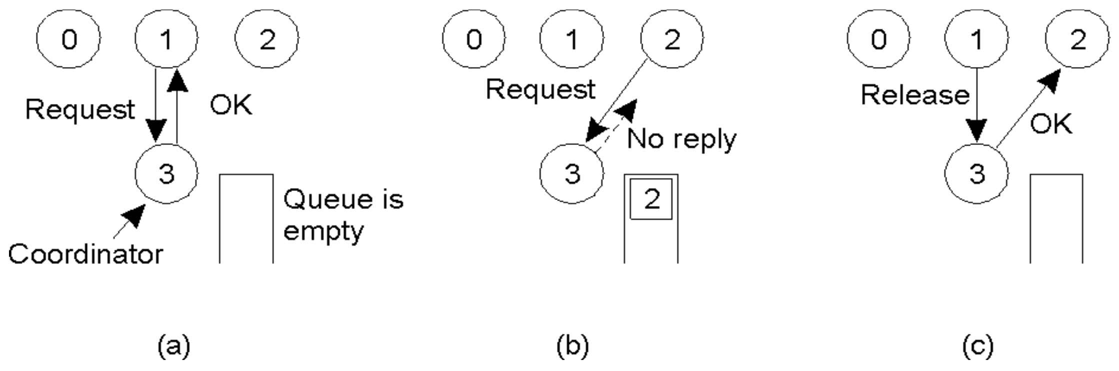


Figure: (a) Process 1 asks the coordinator for permission to enter a critical region. Permission is granted. (b) Process 2 then asks permission to enter the same critical region. The coordinator does not reply. (c) When process 1 exits the critical region, it tells the coordinator, which then replies to 2.

- The most straight forward way to achieve mutual exclusion in a distributed system is to simulate how it is done in a one-processor system.
- One process is elected as the coordinator (e.g., the one running on the machine with the highest network address).
- Whenever a process wants to enter a critical region, it sends a request message to the coordinator stating which critical region it wants to enter and asking for permission.

- If no other process is currently in that critical region, the coordinator sends back a reply granting permission.
- When the reply arrives, the requesting process enters the critical region.
- Now suppose that another process 2 in figure asks for permission to enter the same critical region.
- The coordinator knows that a different process is already in the critical region, so it cannot grant permission.
- The exact method used to deny permission is system dependent.
- In Fig.(b), the coordinator just refrains from replying, thus blocking process 2, which is waiting for a reply.
- Alternatively, it could send a reply saying ''permission denied.'' Either way, it queues the request from 2 for the time being and waits for more messages.
- When process 1 exits the critical region, it sends a message to the coordinator releasing its exclusive access, as shown in Fig. (c).
- The coordinator takes the first item off the queue of deferred requests and sends that process a grant message.
- If the process was still blocked (i.e., this is the first message to it), it unblocks and enters the critical region.
- If an explicit message has already been sent denying permission, the process will have to poll for incoming traffic or block later.
- Either way, when it sees the grant, it can enter the critical region.

Distributed Mutual Exclusion algorithm

- When a process wants to enter a critical region, it builds a message containing the name of the critical region it wants to enter, its process number, and the current time.
- It then sends the message to all other processes, conceptually including itself.
- The sending of messages is assumed to be reliable; that is, every message is acknowledged.
- Reliable group communication if available, can be used instead of individual messages.
- When a process receives a request message from another process, the action it takes depends on its state with respect to the critical region named in the message.
- Three cases have to be distinguished:
    - If the receiver is not in the critical region and does not want to enter it, it sends back an OK message to the sender.
    - If the receiver is already in the critical region, it does not reply. Instead, it queues the request.
    - If the receiver wants to enter the critical region but has not yet done so, it compares the timestamp in the incoming message with the one contained in the message that it has sent everyone.
    - The lowest one wins. If the incoming message is lower, the receiver sends back an OK message.
    - If its own message has a lower timestamp, the receiver queues the incoming request and sends nothing.

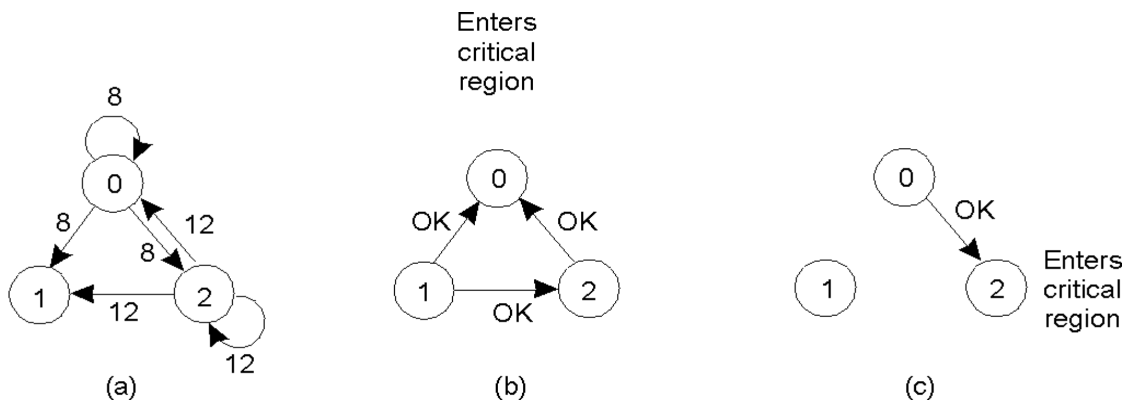


Figure (a) Two processes want to enter the same critical region at the same moment. (b) Process 0 has the lowest timestamp, so it wins. (c) When process 0 is done, it sends an OK also, so 2 can now enter the critical region.

- Process 0 sends everyone a request with timestamp 8.
- while at the same time, process 2 sends everyone a request with timestamp 12.
- Process 1 is not interested in entering the critical region, so it sends OK to both senders.
- Processes 0 and 2 both see the conflict and compare timestamps.
- Process 2 sees that it has lost, so it grants permission to 0 by sending OK.
- Process 0 now queues the request from 2 for later processing and enters the critical region.
- When it is finished, it removes the request from 2 from its queue and sends an OK message to process 2, allowing the latter to enter its critical region.

Token Ring Algorithm in Mutual Exclusion

- Here we have a bus network, as shown in Fig. (a), (e.g., Ethernet), with no inherent ordering of the processes.
- In software, a logical ring is constructed in which each process is assigned a position in the ring, as shown in Fig.(b).
- The ring positions may be allocated in numerical order of network addresses or some other means.
- It does not matter what the ordering is. All that matters is that each process knows who is next in line after itself.

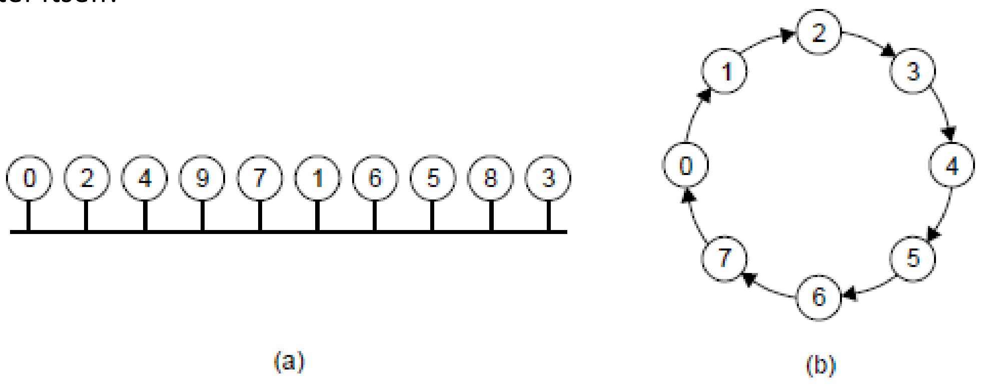


Figure (a) An unordered group of processes on a network. (b) A logical ring constructed in software.

- When the ring is initialized, process 0 is given a token.
- The token circulates around the ring.
- It is passed from process k to process k +1 (modulo the ring size) in point-to-point messages.
- When a process acquires the token from its neighbor, it checks to see if it is attempting to enter a critical region.

- If so, the process enters the region, does all the work it needs to, and leaves the region.
- After it has exited, it passes the token along the ring. It is not permitted to enter a second critical region using the same token.
- If a process is handed the token by its neighbor and is not interested in entering a critical region, it just passes it along.
- As a consequence, when no processes want to enter any critical regions, the token just circulates at high speed around the ring.

Deadlock

- A set of processes is deadlocked if each process in the set is waiting for an event that only another process in the set can cause.
- Suppose process D holds resource T and process C holds resource U.
- Now process D requests resource U and process C requests resource T but none of process will get this resource because these resources are already hold by other process so both can be blocked, with neither one be able to proceed, this situation is called deadlock.
- Example:

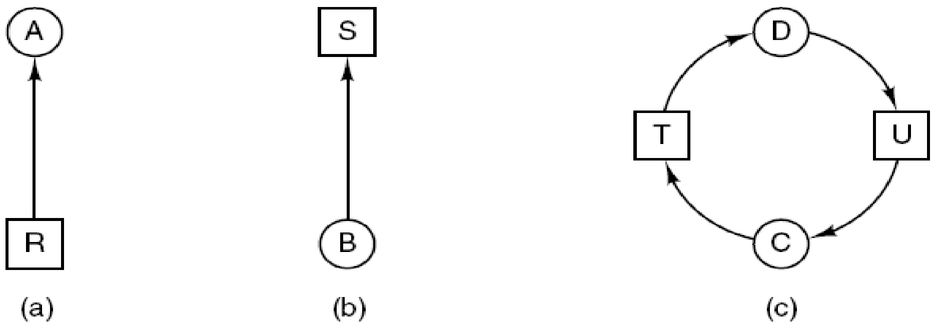


Figure: Resource allocation graphs. (a) Holding a resource. (b) Requesting a resource. (c) Deadlock.

- As shown in above figure, resource T assigned to process D and resource U is assigned to process C.
- Process D is requesting / waiting for resource U and process C is requesti ng / waiting for resource T.

Processes C and D are in deadlock over resources T and U.

Conditions that lead to deadlock

- There are four conditions that must hold for deadlock:

1) Mutual exclusion condition
   - Each resource is either currently assigned to exactly one process or is available.
2) Hold and wait condition
   - Process currently holding resources granted earlier can request more resources.
3) No preemption condition
   - Previously granted resources cannot be forcibly taken away from process.
4) Circular wait condition
   - There must be a circular chain of 2 or more processes. Each process is waiting for resource that is held by next member of the chain.

- **All the four of these conditions must be present for a deadlock to occur.**

Deadlock detection and recovery

Deadlock detection with single resource of each type.

- Algorithm for detecting deadlock for single resource
  1. For each node, N in the graph, perform the following five steps with N as the starting node.
  2. Initialize L to the empty list, designate all arcs as unmarked.
  3. Add current node to end of L, check to see if node now appears in L two times. If it does, graph contains a cycle (listed in L), algorithm terminates.
  4. From given node, see if any unmarked outgoing arcs. If so, go to step 5; if not, go to step 6.
  5. Pick an unmarked outgoing arc at random and mark it. Then follow it to the new current node and go to step 3.
  6. If this is initial node, graph does not contain any cycles, algorithm terminates. Otherwise, dead end. Remove it, go back to previous node, make that one current node, go to step 3.

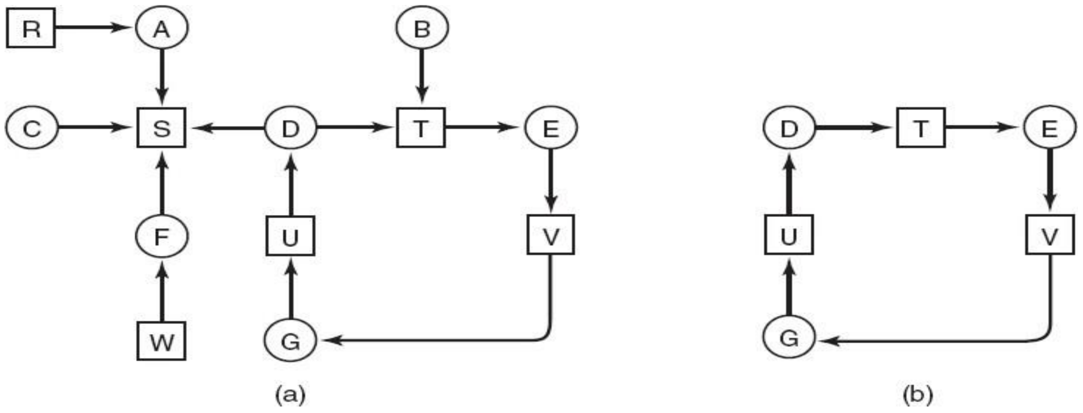


Figure: (a) A resource graph. (b) A cycle extracted from (a)

- For example as shown in the above figure (a),
  - We are starting from node D.
  - Empty list L = ()
  - Add current node so Empty list = (D).
  - From this node there is one outgoing arc to T so add T to empty list.
  - So Empty list become L = (D, T).
  - Continue this step….so we get empty list as below
  - L = (D, T, E)……….. L = (D, T, E, V, G, U, D)
  - In the above step in empty list the node D appears twice, so deadlock.

Deadlock recovery

- **Recovery through preemption**
  (1) In some cases, it may be possible to temporarily take a resource away from its current owner and give it to another process.
  (2) The ability to take a resource away from a process, have another process use it, and then give it back without the process noticing it is highly dependent on the nature of the resource.
  (3) Recovering this way is frequently difficult or impossible.
  (4) Choosing the process to suspend depends largely on which ones have resources that can easily be taken back.
- **Recovery through rollback**
  (1) Create a checkpoint.

(2) Checkpoint a process periodically.
(3) Check pointing a process means that its state is written to a file so that it can be restarted later.
(4) The checkpoint contains not only the memory image, but also the resource state, that is, which resources are currently assigned to the process.
(5) When a deadlock is detected, it is easy to see which resources are needed.
(6) To do the recovery, a process that owns a needed resource is rolled back to a point in time before it acquired some other resource by starting one of its earlier checkpoints.
(7) In effect, the process is reset to an earlier moment when it did not have the resource, which is now assigned to one of the deadlocked processes.
(8) If the restarted process tries to acquire the resource again, it will have to wait until it becomes available.

- **Recovery through killing processes**
  (1) The crudest, but simplest way to break a deadlock is to kill one or more processes.
  (2) One possibility is to kill a process in the cycle. With a little luck, the other processes will be able to continue.
  (3) If this does not help, it can be repeated until the cycle is broken.
  (4) Alternatively, a process not in the cycle can be chosen as the victim in order to release its resources.
  (5) In this approach, the process to be killed is carefully chosen because it is holding resources that some process in the cycle needs.

Deadlock Prevention

- Deadlock can be prevented by attacking the one of the four conditions that leads to deadlock.

1) **Attacking the Mutual Exclusion Condition**
   - No deadlock if no resource is ever assigned exclusively to a single process.
   - Some devices can be spooled such as printer, by spooling printer output; several processes can generate output at the same time.
   - Only the printer daemon process uses physical printer.
   - Thus deadlock for printer can be eliminated.
   - Not all devices can be spooled.
   - Principle:
     - Avoid assigning a resource when that is not absolutely necessary.
     - Try to make sure that as few processes as possible actually claim the resource.
2) **Attacking the Hold and Wait Condition**
   - Require processes to request all their resources before starting execution.
   - A process is allowed to run if all resources it needed is available. Otherwise nothing will be allocated and it will just wait.
   - Problem with this strategy is that a process may not know required resources at start of run.
   - Resource will not be used optimally.
   - It also ties up resources other processes could be using.
   - Variation: A process must give up all resources before making a new request. Process is then granted all prior resources as well as the new ones only if all required resources are available.
   - Problem: what if someone grabs the resources in the meantime how can the processes save its state?
3) **Attacking the No Preemption Condition**
   - This is not a possible option.
   - When a process P0 request some resource R which is held by another process P1 then resource R is forcibly taken away from the process P1 and allocated to P0.

- Consider a process holds the printer, halfway through its job; taking the printer away from this process without having any ill effect is not possible.

4) Attacking the Circular Wait Condition

- To provide a global numbering of all the resources.
- Now the rule is this: processes can request resources whenever they want to, but all requests must be made in numerical order.
- A process need not acquire them all at once.
- Circular wait is prevented if a process holding resource n cannot wait for resource m, if m > n.
- No way to complete a cycle.

Election Algorithms in Distributed Systems

- Several distributed algorithms require a coordinator process in the entire System.
- It performs coordination activity needed for the smooth running of processes.
- Two examples of such coordinator processes encountered here are:
  - The coordinator in the centralized algorithm for mutual exclusion.
  - The central coordinator in the centralized deadlock detection algorithm.
- If the coordinator process fails, a new coordinator process must be elected to take up the job of the failed coordinator.
- Election algorithms are meant for electing a coordinator process from the currently running processes.
- Election algorithms are based on the following assumptions:
  - Each process in the system has a unique priority number.
  - Whenever an election is held, the process having the highest priority number among the currently active processes is elected as the coordinator.
  - On recovery, a failed process can take appropriate actions to rejoin the set of active processes.

Bully Algorithm for Election

- Bully algorithm specifies the process with the highest identifier will be the coordinator of the group. It works as follows:
- When a process p detects that the coordinator is not responding to requests, it initiates an election:
  - p sends an election message to all processes with higher numbers.
  - If nobody responds, then p wins and takes over.
  - If one of the processes answers, then p’s job is done.
- If a process receives an election message from a lower-numbered process at any time, it:
  - sends an OK message back.
  - holds an election (unless its already holding one).
- A process announces its victory by sending all processes a message telling them that it is the new coordinator.
- If a process that has been down recovers, it holds an election.

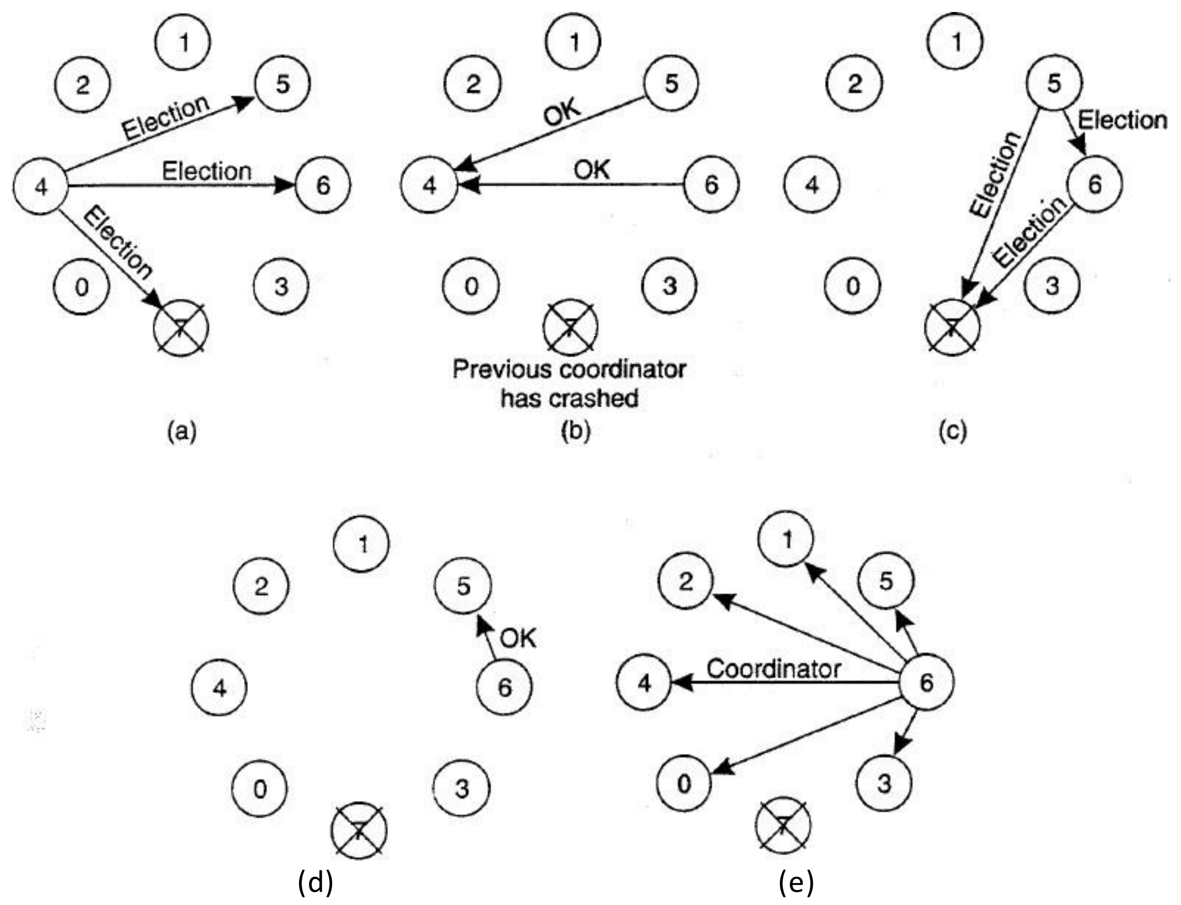
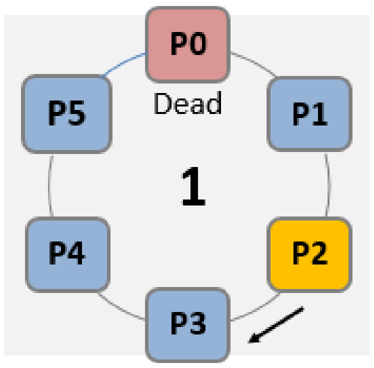


Figure: Bully Algorithm. (a) Process 4 holds an election. (b) Processes 5 and 6 respond, telling 4 to stop. (c) Now 5 and 6 each hold an election. (d) Process 6 tells 5 to stop. (e) Process 6 wins and tells everyone.

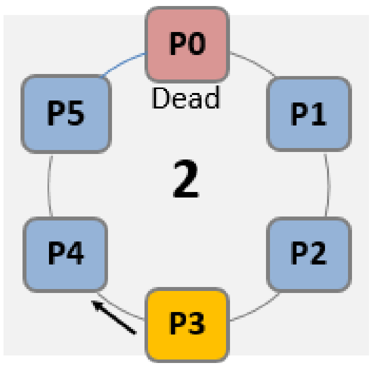
- The group consists of eight processes, numbered from 0 to 7.
- Previously process 7 was the coordinator, but it has just crashed.
- Process 4 is the first one to notice this, so it sends ELECTION messages to all the processes higher than it, namely 5, 6, and 7, as shown in Figure (a).
- Processes 5 and 6 both respond with OK, as shown in Figure (b).
- Upon getting the first of these responses, 4 knows that its job is over.
- It knows that one of these bigwigs will take over and become coordinator.
- It just sits back and waits to see who the winner will be.
- In Figure (c), both 5 and 6 hold elections, each one only sending messages to those processes higher than itself.
- In Figure (d) process 6 tells 5 that it will take over.
- At this point 6 knows that 7 is dead and (6) is the winner.
- If there is state information to be collected from disk or elsewhere to pick up where the old coordinator left off, 6 must now do what is needed.
- When it is ready to take over, 6 announces this by sending a COORDINATOR message to all running processes.
- When 4 gets this message, it can now continue with the operation it was trying to do when it discovered that 7 was dead, but using 6 as the coordinator this time.
- In this way the failure of 7 is handled and the work can continue.
- If process 7 is ever restarted, it will just send all the others a COORDINATOR message and bully them into submission.

Ring Election Algorithm

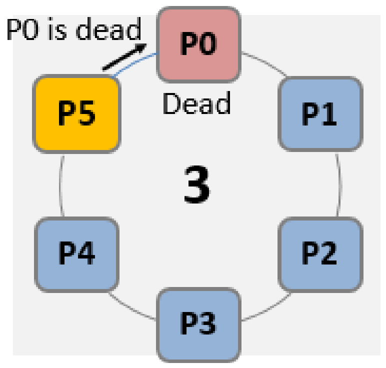
- The ring algorithm assumes that the processes are arranged in a logical ring and each process is knowing the order of the ring of processes.
- If any process detects failure, it constructs an election message with its process ID (e.g., its network address and local process ID) and sends it to its neighbor.
- If the neighbor is down, the process skips over it and sends the message to the next process in the ring.
- This process is repeated until a running process is located.
- At each step, the process adds its own process ID to the list in the message and sends the message to its living neighbor.
- Eventually, the election message comes back to the process that started it.
- The process then picks either the highest or lowest process ID in the list and sends out a message to the group informing them of the new coordinator.



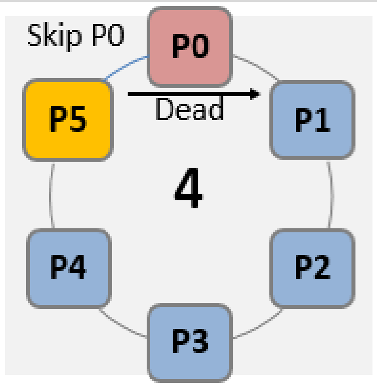
Election:{P2}



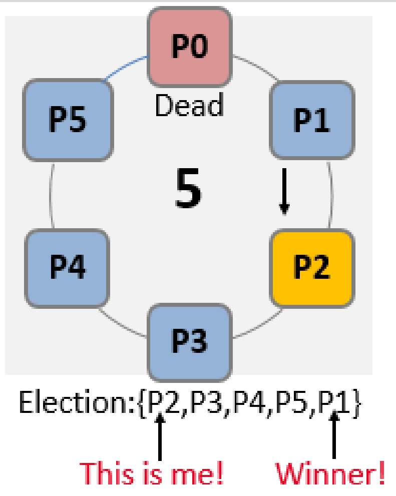
Election:{P2,P3}



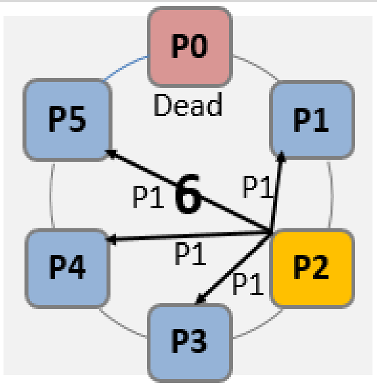
Election:{P2,P3,P4,P5}



Election:{P2,P3,P4,P5}



Election:{P2,P3,P4,P5,P1}



- Fig.1 shows a ring of six processes. P2 detects that the coordinator P0 is dead.
- It starts an election by sending an election message containing its process ID to its neighbor P3. (Fig.1)
- P3 receives an election message and sends an election message to its neighbor with the ID of P3 suffixed to the list. (Fig.2)
- The same sequence of events occurs with P4, which adds its ID to the list it received from P3 and sends an election message to P5.
- P5 then tries to send an election message to P0. (Fig. 3)
- P0 is dead and the message is not delivered, P5 tries again to its neighbor P1. (Fig. 4)
- The message is received by P2, the originator of the election. (Fig. 5)

- P2 recognizes that it is the initiator of the election because its ID is the first in the list of processes.
- It then picks a leader, it chooses the lowest-numbered process ID, which is that of P1.
- It then informs the rest of the group of the new coordinator (Fig. 6).