1. Clock Synchronization

1.1 Physical Clocks

- Every Uniprocessor system needs a timer mechanism to keep track of time for process execution and accounting for the time spent by the process for using various resources like CPU, I/O or even memory.
- In Distributed system, applications will have several processes running on different machines.
- Ideally global clock is required to coordinate all such processes.
- But in real systems, this is not possible.
- Each CPU has its own clock and it is required that all clocks in the system displays the same time.

1.2 Implementing Computer Clocks

- Nearly all computers have a circuit for keeping track of time.
- A computer timer is usually a precisely machined quartz crystal.
- When kept under tension, quartz crystals oscillate at a well-defined frequency that depends on the kind of crystal, how it is cut, and the amount of tension.
- Associated with each crystal are two registers: a counter and a holding register.
- Holding register holds some constant value.
- Counter register is initialized with holding registers value.
- Each oscillation of the crystal decrements the counter by one.
- When the counter gets to zero, an interrupt is generated and the counter is reloaded from the holding register.
- It is possible to program a timer to generate an interrupt 60 times a second, or at any other desired frequency.
- Each interrupt is called one clock tick.
- When the system is booted initially, it usually asks the operator to enter the date and time, which is then converted to the number of ticks after some known starting date and stored in memory.
- At every clock tick, the interrupt service procedure adds one to the time stored in memory.
- With a single computer and a single clock, it does not matter much if this clock is off by a small amount.
- As soon as multiple CPUs are introduced, each with its own clock, the situation changes.
- Although the frequency at which a crystal oscillator runs is usually fairly stable, it is impossible to guarantee that the crystals in different computers all run at exactly the same frequency.
- In practice, when a system has n computers, all n crystals will run at slightly different rates, causing the clocks gradually to get out of sync and give different values when read out.
- This difference in time values is called **clock skew**.

1.3 Drifting of Clocks

- Each machine is assumed to have a timer that causes an interrupt *H* times a second.
- When this timer goes off, the interrupt handler adds 1 to a software clock that keeps track of the number of ticks (interrupts) since some agreed-upon time in the past.

- Let us call the value of this clock *C*.
- More specifically, when the UTC time is t, the value of the clock on machine *p* is *Cp(t).*
- In a perfect world, we would have *Cp(t)=t* for all p and all t.
- In other words, dC/dt ideally should be 1.
- Theoretically, a timer with H = 60 should generate 216,000 ticks per hour.
- In practice, the relative error obtainable with modern timer chips is about 10^{-5}, meaning that a particular machine can get a value in the range 215,998 to 216,002 ticks per hour.
- If there exists some constant such that

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

- The timer can be said to be working within its specification.
- The constant p is specified by the manufacturer and is known as the maximum drift rate.
- Slow, perfect, and fast clocks are shown in Fig. 3.1

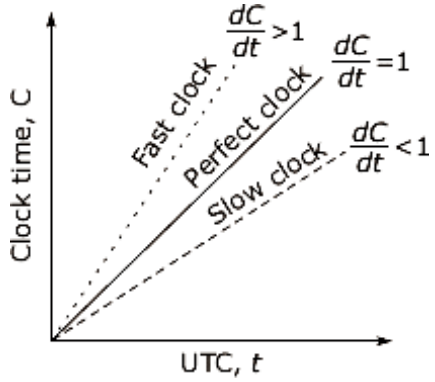


Figure3.1: Drifting of clocks

- If two clocks are drifting from UTC in the opposite direction, at a time *t* after they were synchronized, they may be as much as 2pΔt apart.
- If the operating system designers want to guarantee that no two clocks ever differ by more than δ, clocks must be resynchronized (in software) at least every δ /2 seconds.

2. Clock Synchronization Algorithms

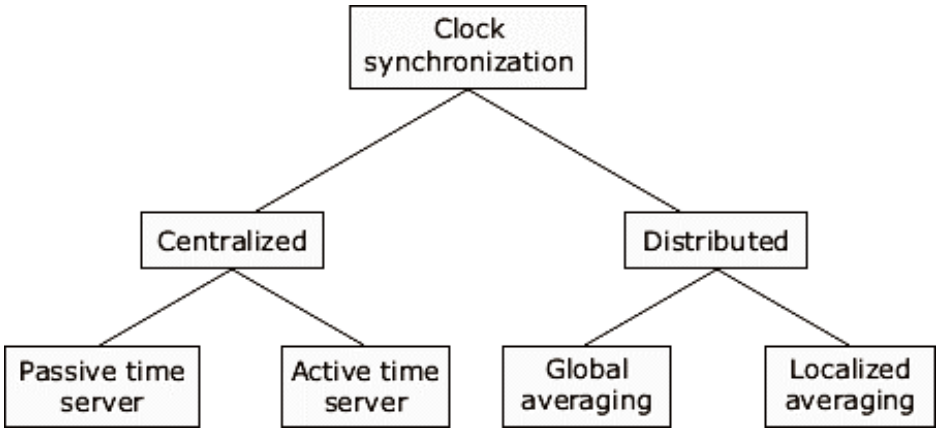


Figure3.2: Classification of Synchronization Algorithms

Centralized Algorithms

- In distributed system using centralized clock synchronization algorithm , one node has a real time receiver (called the time sever node)
- The clock time of this node is correct and used as the reference time.

(a) Passive time server

- 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.

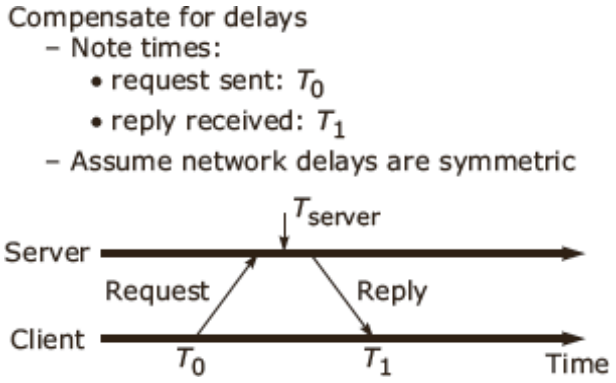


Figure3.3 : Passive time server algorithm

- Assume that client node has a clock time of T_0 when it sends 'time=?' and time T_1 when it receives the 'time=T' message.
- T_0 and T_1 are measured using the same clock , thus the time needed in propagation of message from time server to client node would be $(T_1 - T_0)/2$
- When client node receives the reply from the time server , client node is readjusted to $T_{server}+(T_1-T_0)/2$ as shown in Figure 3.3
- Two methods have been proposed to improve estimated value
  1. 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 prapogation of response message from time server node to client node is taken as $(T_1 - T_0 - I)/2$
     - Clock is adjusted to the value $\boldsymbol{T_{server}+(T_1 - T_0 - I)/2}$

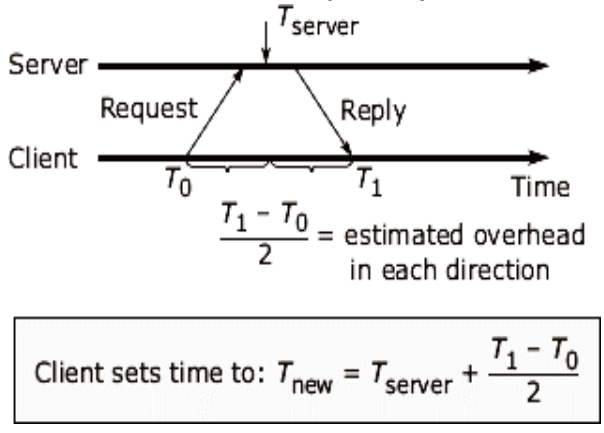
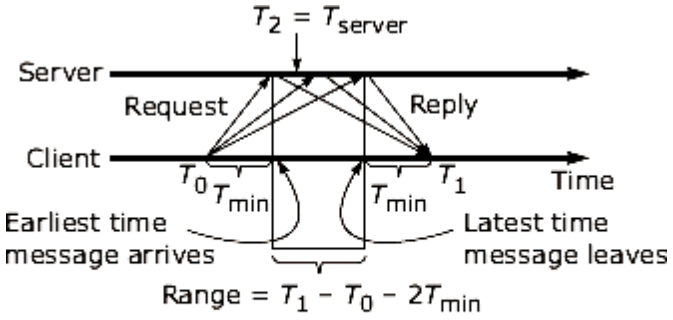


Figure3.4 : Time approximation using passive time server algorithm

2. 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 are 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 (T_1- T_0) is measured many times.



$$\text{Accuracy of result} = \pm \frac{T_1 - T_0}{2} - T_{min}$$

Figure3.5 : Error bounds in Cristian's algorithm

   - Those measurements in which (T_1- T_0) exceeds a specific threshold values are considered to be unreliable and are discarded.
   - Only those values that fall in the range ***(T_1- T_0-2T_{min})*** 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 (T_1- T_0) is minimum , is considered most accurate and half of its value is added to T.

(b) Active time server

- In Cristian's algorithm, the time server is passive.
- Other machines ask it for the time periodically.
- All it does is respond to their queries.

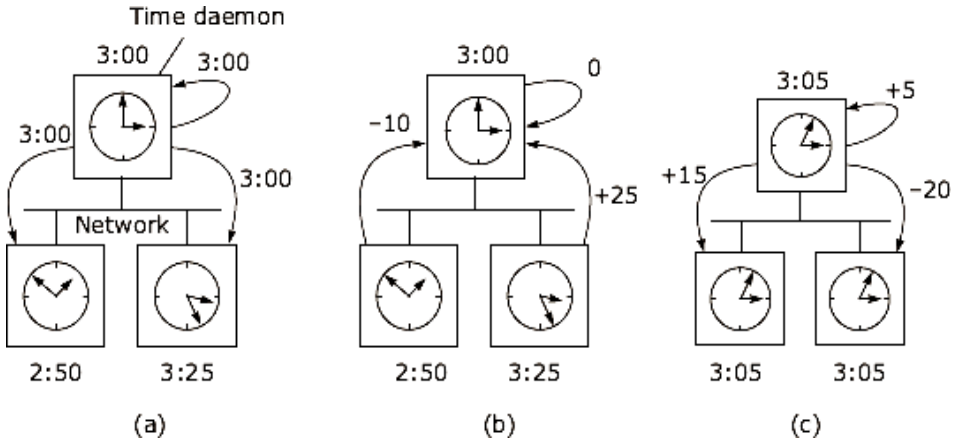


Figure3.6: (a) The time daemon asks all the other machines for their clock values. (b) The machines answer. (c) The time daemon tells everyone how to adjust their clock.

- 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 Figure3.6(a), at 3:00, the time daemon tells the other machines its time and asks for theirs.
- In Figure3.6 (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 Figure3.6 (c).

Distributed Algorithms

(a) Global Averaging algorithm

- One class of decentralized clock synchronization algorithms works by dividing time into fixed-length resynchronization intervals.
- The i^{th} interval starts at T_0+iR and runs until $\mathbf{T_0+(i+1)}\ \boldsymbol{R}$, where T_0 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 defence 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 echoed.

(b) Localized averaging distributed algorithm

- The nodes of the distributed system are logically arranged in a specific pattern like a ring or grid.
- Periodically each node exchanges its clock time with its neighbours in the logical pattern and then , sets its clock time to the average of its own clock time and the clock time of its neighbours.

2.1 Logical Clocks

- Lamport pointed out that clock synchronization need not be absolute.

- If two processes do not interact, it is not necessary that their clocks be synchronized because the lack of synchronization would not be observable and thus could not cause problems.
- Furthermore, he pointed out that what usually matters is not that all processes agree on exactly what time it is, but rather, that they agree on the order in which events occur.
- To synchronize logical clocks, Lamport defined a relation called happens-before.
- The expression *a->b* is read "a happens before *b*" and means that all processes agree that first event *a* occurs, then afterward, event *b* occurs.
- 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->b* is true.
    - 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->b* is also true.
    - Happens –before is a transitive relation, so if a->b and b->c, then a->c.
- If two events, x and y, happen in different processes that do not exchange messages, then x->y is not true, but neither is y->x.
- These events are said to be concurrent, which simply means that nothing can be said about when they happened or which is first.

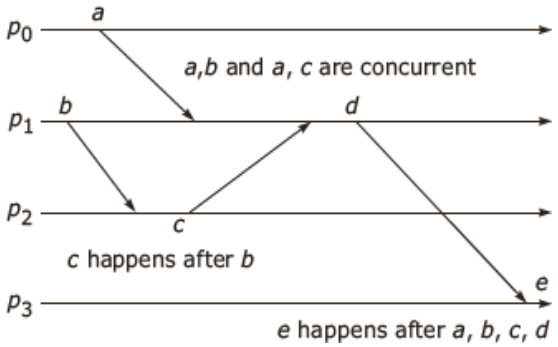


Figure3.7 : Happened before relationship and concurrent events

- We need a way of measuring time such that for every event, *a,* we can assign it a time value *C(a)* on which all processes agree.
- Consider the three processes depicted in Fig. 3.8(a).
- The processes run on different machines, each with its own clock, running at its own speed.

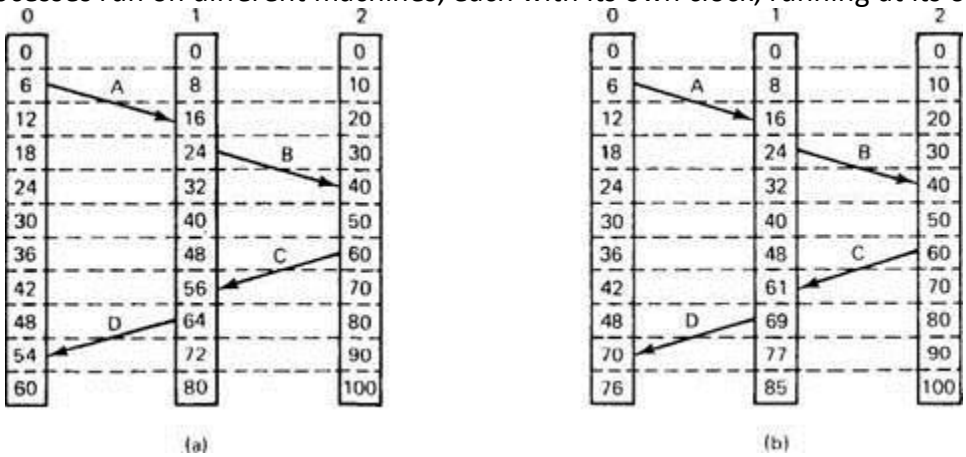


Figure3.8: (a) Three processes, each with its own clock. The clocks run at different rates. (b) Lamport's algorithm corrects the clocks.

- As can be seen from the figure, when the clock has ticked 6 times in process 0, it has ticked 8 times in process 1 and 10 times in process 2.
- Each clock runs at a constant rate, but the rates are different due to differences in the crystals.
- At time 6, process 0 sends message *A* to process 1.
- In any event, the clock in process 1 reads 16 when it arrives.
- If the message carries the starting time, 6, in it, process 1 will conclude that it took 10 ticks to make the journey.
- According to this reasoning, message *B* from 1 to 2 takes 16 ticks.
- Message *C* from 2 to 1 leaves at 60 and arrives at 56.
- Similarly, message *D* from 1 to 0 leaves at 64 and arrives at 54.
- These values are clearly impossible hence this situation that must be prevented.
- Lamport's solution, since *C* left at 60, it must arrive at 61 or later.
- Therefore, each message carries the sending time, according to the sender's clock.
- When a message arrives and the receiver's clock shows a value prior to the time the message was sent, the receiver fast forwards its clock to be one more than the sending time.
- In Figure 3.8(b) we see that C now arrives at 61.
- Similarly, *D* arrives at 70.
- Using this method, we now have a way to assign time to all events in a distributed system subject to the following conditions:
    1. If a happens before *b* in the same process, C(a) < C(b).
    2. If *a* and *b* represent the sending and receiving of a message, C(a) < C(b).
    3. For all events *a* and *b,* C(a) ≠C(b).

3. Mutual Exclusion

- Systems involving multiple processes are often most easily programmed using critical regions.
- When a process has to read or update certain shared data structures, it first enters a critical region to achieve mutual exclusion and ensure that no other process will use the shared data structures at the same time.
- In single-processor systems, critical regions are protected using semaphores, monitors, and similar constructs.
- Following are the algorithm used for mutual exclusion

3.1 A Centralized algorithm

- The most straightforward 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.
- 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, as shown in Figure. 3.9(a).
- When the reply arrives, the requesting process enters the critical region.

- Now suppose that another process, 2 in Figure. 3.9(b), 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 Figure. 3.9(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.
- When process 1 exits the critical region, it sends a message to the coordinator releasing its exclusive access, as shown in Figure. 3.9(c).

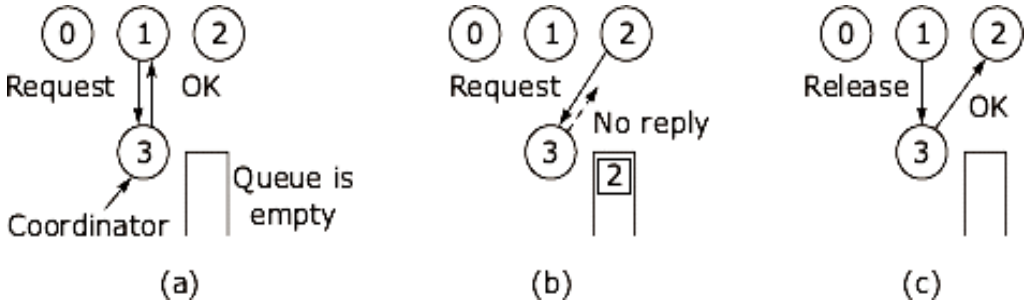


Figure3.9: (a) Process 1 asks the coordinator for permission to enter a critical region. Permission is granted. (b) Process 2 then asks permission to enter the same critical region. The coordinator does not reply. (c) When process 1 exits the critical region, it tells the coordinator, which then replies to 2.

- The coordinator takes the first item off the queue of deferred requests and sends that process a grant message.
- If the process was still blocked, 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.
- The centralized approach also has shortcomings.
- The coordinator is a single point of failure, so if it crashes, the entire system may go down.
- If processes normally block after making a request, they cannot distinguish a dead coordinator from "permission denied" since in both cases no message comes back.
- In addition, in a large system, a single coordinator can become a performance bottleneck.

3.2 A Distributed Algorithm / Ricart and Agrawala's algorithm

- Ricart and Agrawala's algorithm stated that there be a total ordering of all events in the system.
- That is, for any pair of events, such as messages, it must be unambiguous which one happened first.
- 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.

- 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.
  - After sending out requests asking permission to enter a critical region, a process sits back and waits until everyone else has given permission.
  - As soon as all the permissions are in, it may enter the critical region.
  - When it exits the critical region, it sends OK messages to all processes on its queue and deletes them all from the queue.

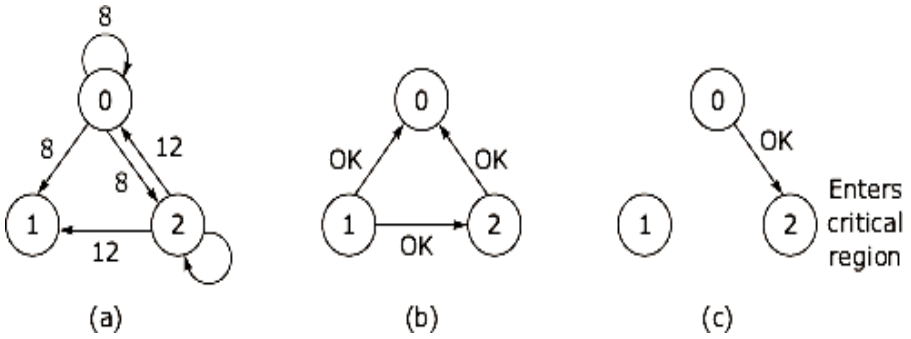


Figure3.10 (a) Two processes want to enter the same critical region at the same moment. (b) Process 0 has the lowest timestamp, so it wins. (c) When process 0 is done, it sends an *OK* also, so 2 can now enter the critical region.

- Suppose that two processes try to enter the same critical region simultaneously, as shown in Fig. 3.10(a).
- 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, as shown in Fig. 3.10(b).
- 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, as shown in Figure. 3.10(c).
- If any process crashes, it will fail to respond to requests.

- This silence will be interpreted as denial of permission, thus blocking all subsequent attempts by all processes to enter all critical regions.
- Solution to this is when a request comes in, the receiver always sends a reply, either granting or denying permission.
- Whenever either a request or a reply is lost, the sender times out and keeps trying until either a reply comes back or the sender concludes that the destination is dead.
- After a request is denied, the sender should block waiting for a subsequent OK message.

3.3 Token Ring Algorithm

- We have a bus network, as shown in Fig. 3.11(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. 3.11(b).
- Each process should know who is next in line after itself.

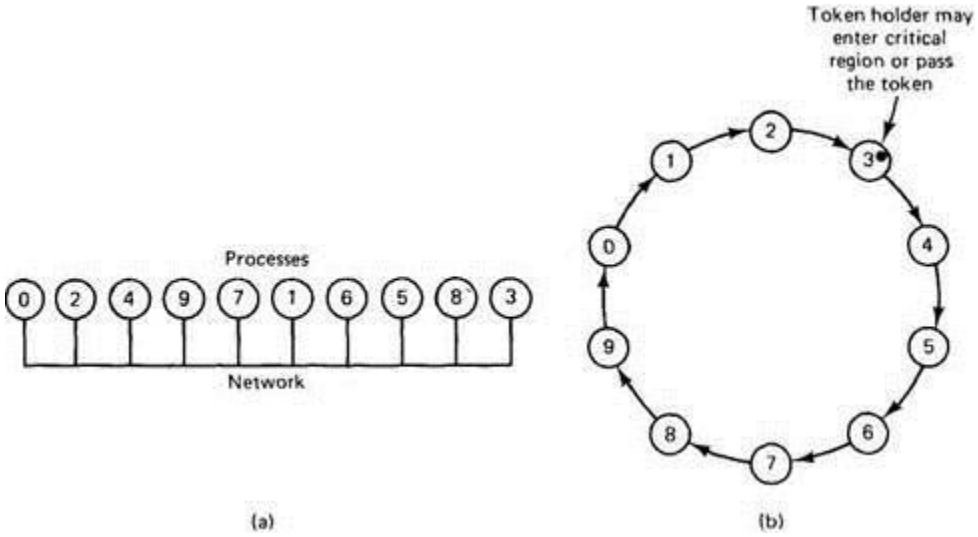


Figure 3.11 (a) An unordered group of processes on a network. (b) A logical ring constructed in software.

- 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 neighbour, 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 neighbour 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.
- Only one process has the token at any instant, so only one process can be in a critical region.

- Since the token circulates among the processes in a well-defined order, starvation cannot occur.

3.4 Comparison of Various Algorithms

Algorithm	Messages per entry/exit	Delay before entry (in messages times)	Problems
Centralized	3	2	Coordinator crash
Distributed	2(n-1)	2(n-1)	Crash of any process
Token Ring	1 to ∞	0 to n-1	Lost token, process crash

Table3.1: Comparison of all Algorithm

4. Election Algorithm

4.1 Bully Algorithm

- When a process notices that the coordinator is no longer responding to requests, it initiates an election. A process, P, holds an election as follows:
  - *P* sends an ELECTION message to all processes with higher numbers.
  - If no one responds, *P* wins the election and becomes coordinator.
  - If one of the higher-ups answers, it takes over. *P*'s job is done.
- At any moment, a process can get an ELECTION message from one of its lower-numbered colleagues.
- When such a message arrives, the receiver sends an OK message back to the sender to indicate that he is alive and will take over.
- The receiver then holds an election, unless it is already holding one.
- Eventually, all processes give up but one, and that one is the new coordinator.
- It announces its victory by sending all processes a message telling them that starting immediately it is the new coordinator.
- Thus the biggest guy in town always wins, hence the name "bully algorithm."

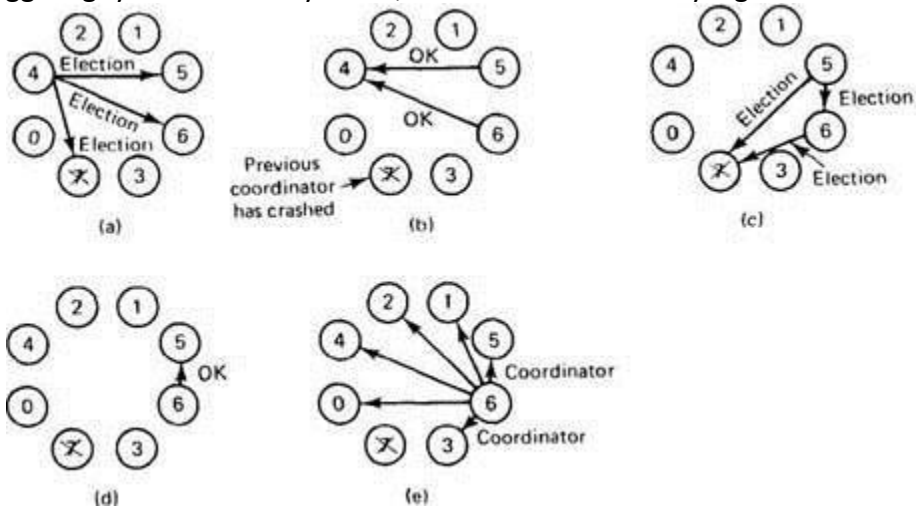


Figure3.12: Bully Algorithm.(a) Process 4 holds an election. (b) Processes 5 and 6 respond, telling 4 to stop. (c) Now 5 and 6 each hold an election. (d) Process 6 tells 5 to stop. (e) Process 6 wins and tells everyone.

- In Figure 3.12 we see how the bully algorithm works.
- 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 3.12(a).
- Processes 5 and 6 both respond with OK, as shown in Figure 3.12 (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 3.12 (c), both 5 and 6 hold elections, each one only sending messages to those processes higher than itself.
- In Figure 3.12 (d) process 6 tells 5 that it will take over.
- At this point 6 knows that 7 is dead and that it (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.

4.2 A Ring Algorithm

- Another election algorithm is based on the use of a ring, but without a token.
- We assume that the processes are physically or logically ordered, so that each process knows who its successor is.
- When any process notices that the coordinator is not functioning, it builds an ELECTION message containing its own process number and sends the message to its successor.
- If the successor is down, the sender skips over the successor and goes to the next member along the ring, or the one after that, until a running process is located.
- At each step, the sender adds its own process number to the list in the message.
- Eventually, the message gets back to the process that started it all.
- That process recognizes this event when it receives an incoming message containing its own process number.
- At that point, the message type is changed to COORDINATOR and circulated once again, this time to inform everyone else who the coordinator is, the list member with the highest number and who the members of the new ring are.
- When this message has circulated once, it is removed and everyone goes back to work.

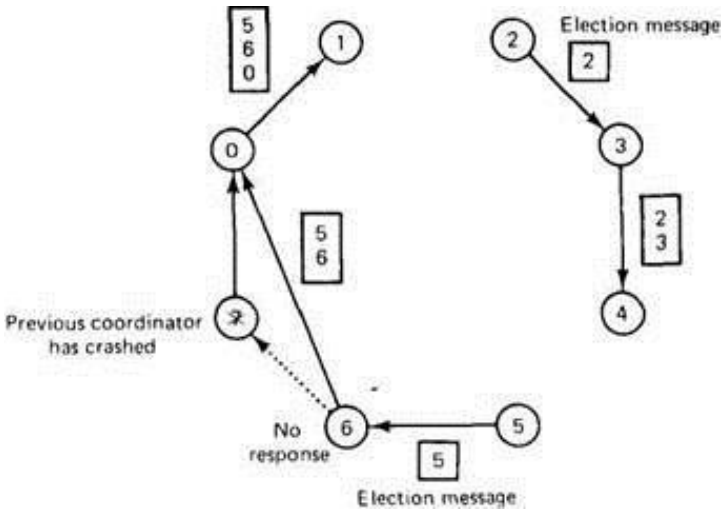


Figure3.13: Election algorithm using a ring.

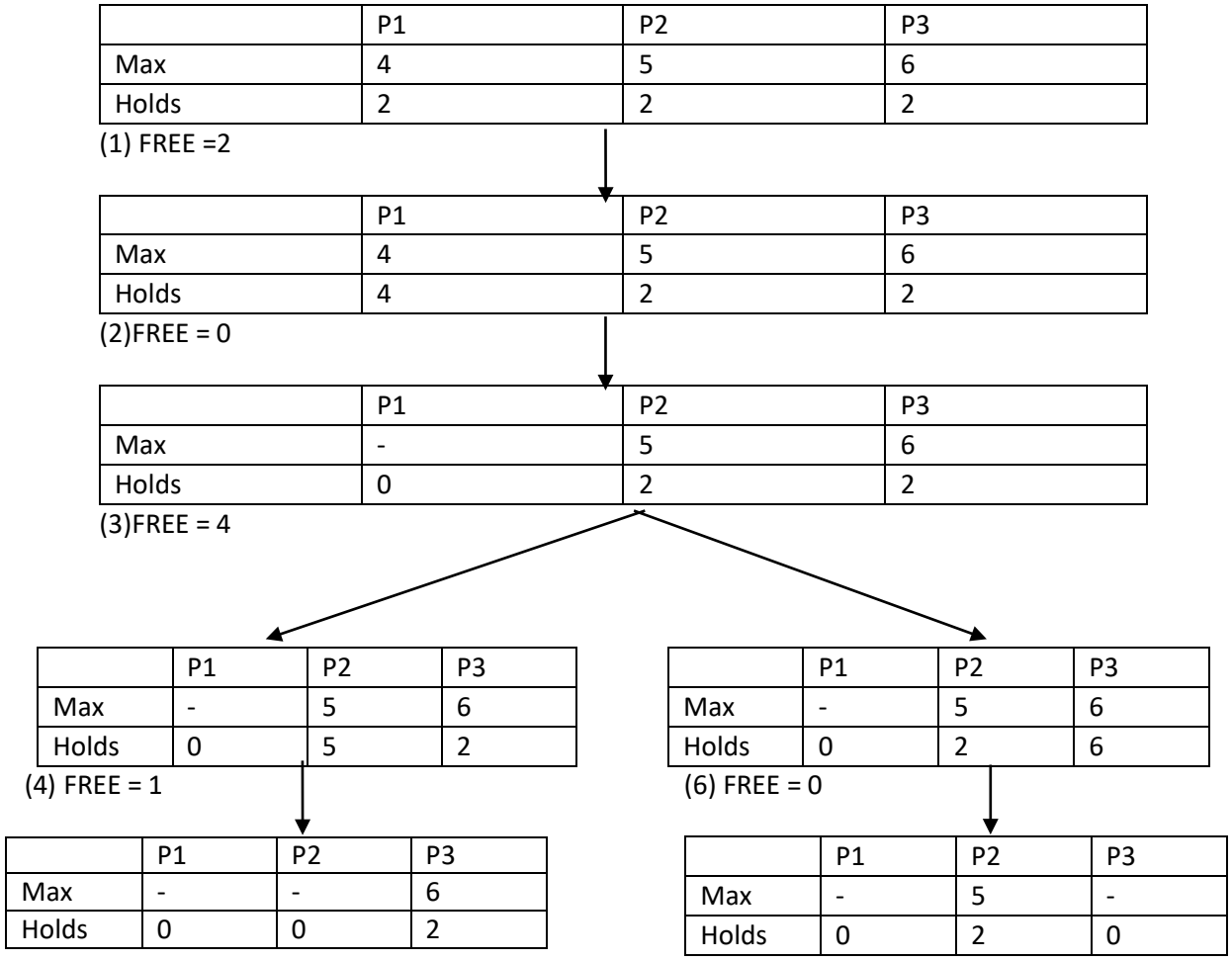
- In Figure 3.13 we see what happens if two processes, 2 and 5, discover simultaneously that the previous coordinator, process 7, has crashed.
- Each of these builds an ELECTION message and starts circulating it.
- Eventually, both messages will go all the way around, and both 2 and 5 will convert them into COORDINATOR messages, with exactly the same members and in the same order.
- When both have gone around again, both will be removed.
- It does no harm to have extra messages circulating; at most it wastes a little bandwidth

5. Deadlocks in Distributed System

- Deadlocks in distributed systems are similar to deadlocks in single-processor systems.
- They are harder to avoid, prevent, or even detect, and harder to cure when tracked down because all the relevant information is scattered over many machines.
- Various strategies are used to handle deadlocks.
- Four of the best-known ones are listed below
  1. The ostrich algorithm (ignore the problem).
  2. Avoidance (avoid deadlocks by allocating resources carefully).
  3. Prevention (statically make deadlocks structurally impossible).
  4. Detection (let deadlocks occur, detect them, and try to recover).

5.1 Deadlock Avoidance

- Deadlock avoidance techniques use some advance knowledge of the resource usage of processes to predict the future state of the system for avoiding allocation that can generally lead the deadlock.
- Deadlock avoidance algorithms are usually work in the following steps:
- When process request for resources, even resource is available for allocation, resource is not immediately allocated to the process.



- Advance knowledge of the resource usage of processes, the systems perform some analysis to decide whether granting the process's request is safe or unsafe.
- Resource is allocated to the process when the analysis shows that it is safe to do well; otherwise request is deferred.
- Deadlock avoidance algorithms are based on the concept of safe and unsafe states.
- A system is said to be in safe state if it is not in a deadlock state and there exists some ordering of them to completion.
- Any ordering of the processes that can guarantee the completion of all the processes is called safe sequence.
- The concept of safe and unsafe states can be best illustrated with the help of an example.
- Let us assume that system there are total of 8 units of a particular resource type for three processes P1. P2 and P3 are competing.
- For this state there are two safe sequences (P1, P2, P3) and (P1, P3, P2).
- Starting from the state of fig (1) the scheduler could simply run P1 exclusively until it asked for & got two more units of the resources that are currently free leading to the state of figure (2) when P1 complete a &release the resources held by it.

- We get the state of figure (3) then scheduler chooses to run P2 eventually leading to the state of figure (4) when P2 complete and releases the resources held by it, system enter the state of figure (5) now available resources P3 can be run to complete.
- The initial state of figure (a) is safe state because the system by careful scheduling can avoid deadlock.
- This example also shows that for a particular state there may be more than one safe sequence.
- If resource allocation is not done, the system may move from safe state to an unsafe state.
- Let us consider the example shown Figure 3.14(b).
- Figure (8) is same initial state as that of Figure 3.14(a).
- This time suppose process P2 request for one additional unit of the resource free and the same is allocated to it by the system.

	P1	P2	P3
Max	4	5	6
Holds	2	2	2

(8) FREE = 2

	P1	P2	P3
Max	4	5	6
Holds	2	3	2

(9) FREE = 1

Figure3.14 (b) :Unsafe Sequence

- Then resulting system state is shown in figure (i) is no longer safe state because we have only one unit of the resource free.
- Deadlock avoidance algorithm performs resource allocation in such a way as to ensure that the system will always remain in safe state.

5.2 Deadlock Prevention.

(a) Collective request

- This method ensures the hold and wait condition by ensuring that whenever a process request for resource, it does not hold any other resource.
- Also a process must collect all its resource before it commences execution.
- There are mainly two policies:
  1. A process must request all its resource before execution begins.
     - If all resources are available, they are allocated and process runs to completion.
     - If any one of the requested resource is not available , none will be allocated and process has to wait
  2. A process can request resource during execution, with the rule that it can request resources only if it does not hold any other resources.
     - If the process is holding some resource, it adheres to this rule by releasing the resources hold and re-requesting the necessary resources.

(b) Ordered request

- In the circular wait method, each resource type is assigned a unique global number to ensure a total ordering of all resource types.
- The resource allocation policy is such that a process can request a resource at any time, but the process must not request a resource with a number lower than the number of any of the resource that it is already holding.
- For example, if a process holds a resource type *i,* it may request a resource type having the number *j* only if *j>i.*
- If a process needs several units of the same resource type, it may issue a single request for all the units.

(c) Preemption

- A pre-emptable resource is a resource whose state can be saved and restored later.
- This implies that the resource can be taken away temporarily from the process to which it is currently allocated without disturbing the computation performed so far.
- There are two algorithms for pre-emption:

1. Wait and die algorithm

- As shown in figure there are two possibility
- (a): an old process wants a resource held by a young process.
- (b): a young process wants a resource held by an old process.

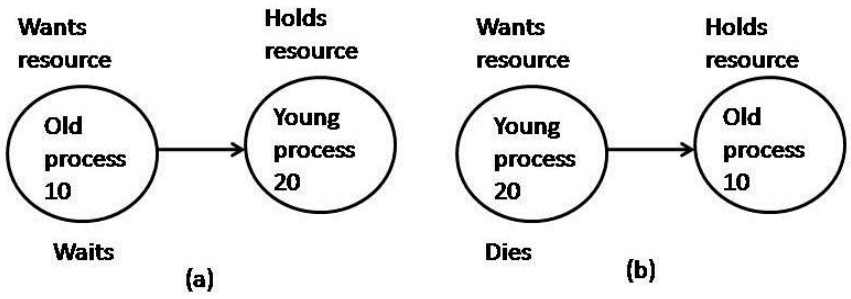


Figure3.15: Wait and Die algorithm

- In one case we should allow the process to wait; in the other we should kill it.
- Suppose that we label (a) dies and (b) wait.
- Then we are killing off an old process trying to use a resource held by a young process, which is inefficient.
- Under these conditions, the arrows always point in the direction of increasing transaction numbers, making cycles impossible.
- This algorithm is called *wait-die*.

2. Wound and wait algorithm

- This algorithm allows pre-emption.
- A young whippersnapper will not pre-empt a venerable old one.
- One transaction is supposedly wounded (it is actually killed) and the other waits.
- If an old process wants a resource held by a young one, the old process pre-empts the young one, whose transaction is then killed.
- The young one probably starts up again immediately, and tries to acquire the resource, forcing it to wait.

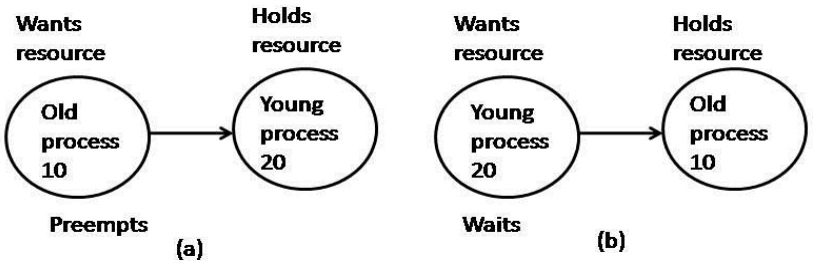


Figure3.16: Wound and Wait algorithm

- If an old timer wants a resource held by a young one, the old timer waits politely.
- If the young one wants a resource held by the old one, the young one is killed.
- It will undoubtedly start up again and be killed again.
- This cycle may go on many times before the old one release the resource.

5.3 Deadlock Detection

- When a deadlock is detected in a conventional OS, the way to resolve sit is to kill off one or more processes.
- Doing so invariably leads to one or more unhappy users.
- When a deadlock is detected in a system based on atomic transactions, it is resolved by aborting one or more transactions.
- Following conditions are sufficient and necessary for a deadlock to occur.
  1. Mutual exclusion condition
     - Each resource is assigned to only one process or is available.
  2. Hold and wait condition
     - Process holding the resource can request additional resources without releasing the resources which are currently held.
  3. No preemption condition
     - Previously granted resources cannot be forcibly taken away.
  4. Circular wait condition
     - Must be a circular chain of two or more processes
     - Each is waiting for a resource held by the next member of the chain.
- Following are the algorithms used to detect deadlocks.

(a) Centralized Deadlock Detection

- This algorithm tries to imitate the non-distributed algorithm.
- Each machine maintains the resource graph for its own processes & resources.
- A central coordinator maintains the resource graph for the entire system.
- Whenever an arc is added or deleted from the resource graph, a message can be sent to the coordinator providing the update.
- Periodically, every process can send a list of arcs added or deleted since the previous update – requires fewer messages than the first one.

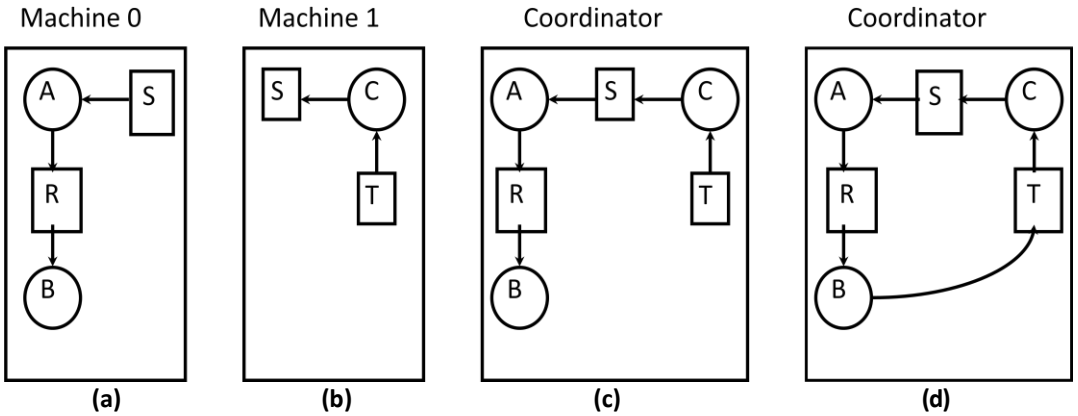


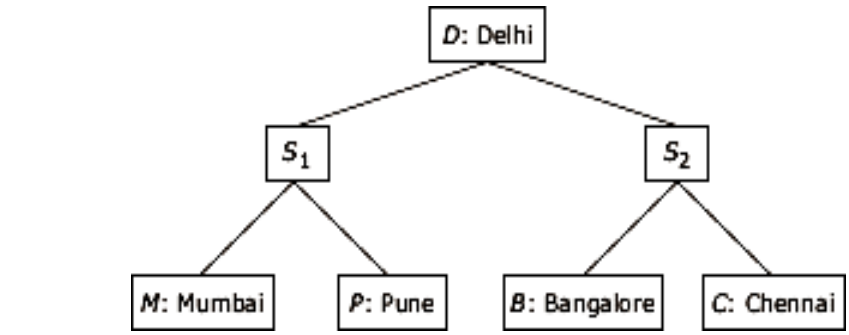
Figure3.17: Centralized Deadlock Detection.

- The coordinator can ask for information when it needs it.
- When the coordinator detects a cycle, it kills off one process to break the deadlock.
- Consider a system with processes A and B running on machine 0, and process C on machine 1.
- Three resources exist: R, S and T.
- Initially: A holds S but wants R, which it cannot have because B is using it; C has T and wants S, too.
- The coordinator's view of the world is shown in Figure 3.17(c).
- This configuration is safe: as soon as B finishes, A can get R and finish, releasing S for C.
- After a while B releases R and asks for T, a perfectly legal and safe swap.
- Machine 0 sends a message to the coordinator announcing the release of R.
- Machine 1 sends a message to the coordinator announcing the fact that B is now waiting for its resource, T.
- Unfortunately, the message from machine 1 arrives first, leading the coordinator to construct the graph of (d).
- The coordinator incorrectly concludes that a deadlock exists and kills some process.
- Such a situation is called a *false deadlock*.

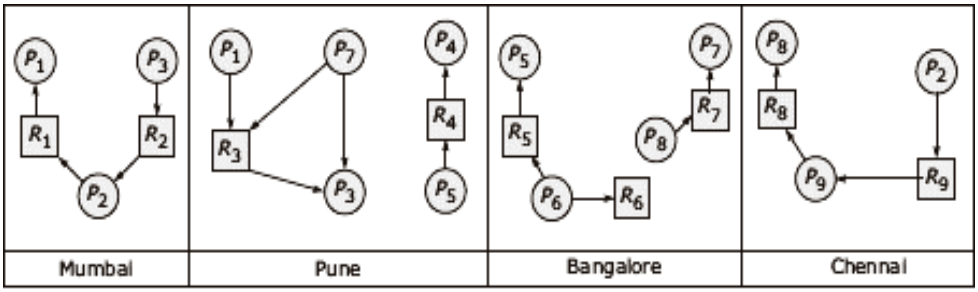
(b) Hierarchical Control

- The hierarchical approach uses logical tree of deadlock detectors called controllers.
- Each controller is responsible for detecting only those deadlock which involve the sites within its range in the hierarchy.
- Each site has its own local controller which maintains its own local graph.
- Each controller which forms a leaf of the hierarchy maintains the local WFG of a single site.
- Each non- leaf controller maintains a WFG which is the union of WFGs of its immediate children in hierarchy.
- Consider a scenario where a distributed system is spread across the entire country as shown in Figure 3.18(a)
- There are four cities: Mumbai, Pune, Chennai and Bangalore in states Maharashtra, Karnataka and Tamil Nadu.
- There are four controllers S_1, S_2, S_3, and S_4 situated at the four cities.

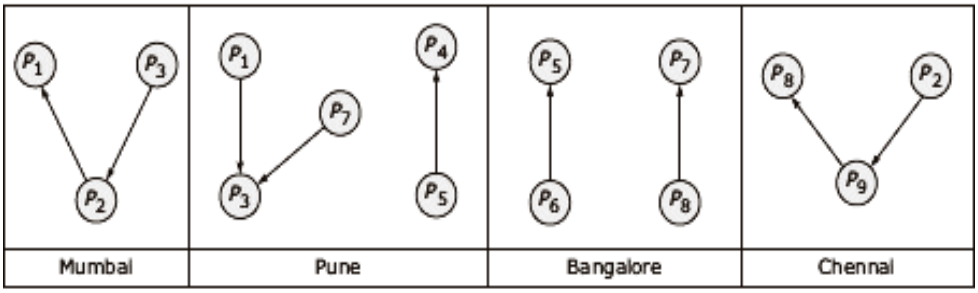
- All four controllers have local WFGs which take necessary action to resolve deadlocks, if any.



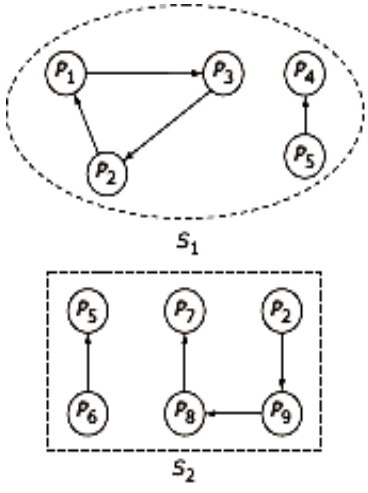
(a) Example of distributed system



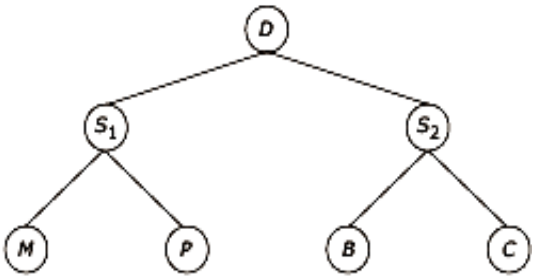
(b) Local RAGs



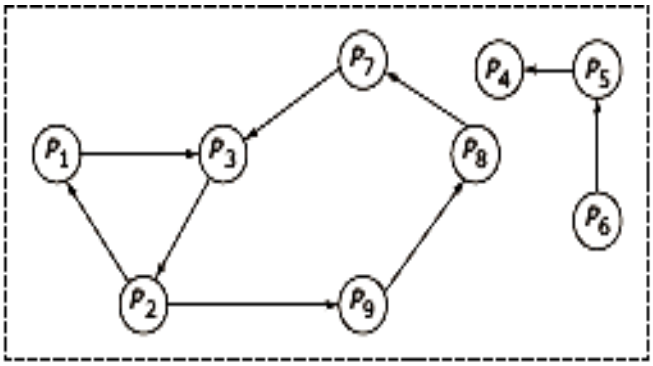
(c) Local WFGs



(d) Global WFG at S_1 and S_2



(e) Location-wise hierarchy of controllers



(f) Deadlock is detected at the last site with the controller at *D*

Figure3.18: Hierarchical deadlock detection

- The WFGs are then sent to the state level controller and state level WFG are combined to give a global WFG.
- As shown in Figure3.18 (e) M, P, B and C maintain WFGs at Mumbai, Pune, Bangalore and Chennai.
- S_1 controls entire Maharashtra and S_2 controls Karnataka and Tamil Nadu.
- The global WFG is combined at Delhi.
- There are two resources R_1 and R_2 at Mumbai, R_3 and R_4 at Pune R_5, R_6 and R_7 at Bangalore and R_8 and R_9 at Chennai.
- Processes P_1, P_2, P_3, P_4, P_5, P_6, P_7, P_8, and P_9 are requesting resources.
- Figure 3.18(f) shows that a deadlock cycle is detected at D.

(c) Distributed Deadlock Detection: Chandy Misra Hass Algorithm.

- In this algorithm processes are allowed to request multiple resources (e.g. locks) at once, instead of one at a time.
- By allowing multiple requests simultaneously, the growing phase of a transaction can be speeded up considerably.
- The consequence of this change to the model is that a process may now wait on two or more resources simultaneously.
- Figure shows a modified resource graph, where only the processes are shown.
- Each arc passes through a resource.
- For simplicity the resources have been omitted from the Figure 3.19

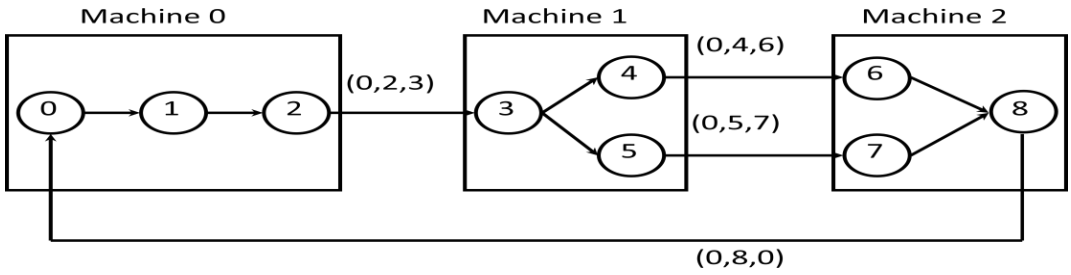


Figure3.19: Chandy Misra Hass Algorithm.

- Process 3 on machine 1 is waiting for two resources, one held by process 4 and one held by process 5.

- Some of the processes are waiting for local resources, such as process 1, but others, such are process 2, are waiting for resources that are located on a different machine.
- These cross-machine arcs that make looking for cycles difficult.
- Algorithm is invoked when a process has to wait for some resource, for example, process 0 blocking on process 1.
- A special probe message is generated and sent to the process (or processes) holding the needed resources.
- The message consists of three numbers: the process that just blocked, the process sending the message, and the process to whom it is being sent.
- The initial message from to 1 contains the triple (0, 0, 1).
- When the message arrives, the recipient checks to see if it itself is waiting for any processes.
- If so, the message is updated, keeping the first field but replacing the second field by its own process number and the third one by the number of the process it is waiting for.
- The message is then sent to the process on which it is blocked.
- If it is blocked on multiple processes, all of them are sent (different) messages.
- Remote messages labelled (0, 2, 3), (0, 4, 6), (0, 5, 7), and (0, 8, 0).
- If a message goes all the way around and comes back to the original sender, that is, the process listed in the first field, a cycle exists and the system is deadlocked.

5.4 Distributed Deadlock Recovery

- Following are the techniques that can be used for deadlock recovery.

(a) Recovery through pre-emption

- The simplest way is to inform the operator that a deadlock has occurred and the processes involved in the deadlock.
- The next step would be to take a resource from some other process and handle the deadlock.
- This depends on nature of resource.
- One option is to abort all deadlocked processes and the other is to abort one process at a time until the deadlock cycle is eliminated.

(b) Recovery through Rollback

- To break a deadlock, it is sufficient to reclaim the resources from the processes which were selected for being killed.
- In order to reclaim the resources, rollback the process to a point where the resource was not allocated to the process.
- The processes are check pointed periodically, and process state is written to a file at regular intervals.
- If required, the process can be restarted from any of the check points when a deadlock is detected and the process is rolled backed to a state as much as is necessary to break the deadlock and the necessary resources can be claimed.

(c) Recovery through killing process

- It involves killing one of the processes in deadlock cycle, reclaiming the resource and reallocating them.

- Deadlock recovery algorithms analyses the resource requirement and interdependencies of the processes involved in the deadlock cycle and select the set of process which if killed can break the cycle.
- The process are chosen such that they can be rerun from the beginning.

5.5 Issues in Recovery from Deadlock

(a) Selection of victims

- Deadlock can be broken by killing or rolling back one of the processes, called victims.
- A victim can be selected based on two factors: minimization of recovery costs and prevention of starvation.

(b) Minimization of recovery cost

- Each system should determine their own cost function to select victims.
- Few factors which may be considered are: the priority of processes, the nature of processes, whether possibility of a rerun will have any ill effects, number and types of resources held by processes, the length of service already received, total number of processes which will be affected.

(c) Prevention of starvation

- If the system objective is only to minimize the recovery cost, it is possible that the same process will be repeatedly selected as a victim and may never go to completion.
- This situation is called starvation and needs to be prevented.
- One approach which may be adopted to avoid starvation is to raise the priority of the process every time is selected as victim.
- Other include the number of times a process is victimized as a parameter in the cost function and then select the victim.

(d) Use of transaction mechanism

- After a process is killed or rolled back for recovery from a deadlock, it needs to be rerun.
- This may not always be safe, because it may have performed some non-idempotent operations.
- If a process has updated the amount in a bank account by a credit operation, re-execution will result in one more credit and an incorrect state.

1. Threads