TOPIC 6

SYNCHRONIZATION

Introduction

Synchronization mechanisms that are suitable for distributed systems. In particular, the following synchronization related issues are described:
Clock synchronization Event ordering
Mutual exclusion Deadlock
Election algorithm

Structure:

6.1 Clock Synchronization
6.2 How Computer Clocks are implemented
6.3 Mutual Exclusion
6.4 Election Algorithms
6.5 Mutual Exclusion

Learning Activity 6.0

1. Explain Mutual Exclusion
2. Discuss Clock Synchronization

6.1 CLOCK SYNCHRONIZATION

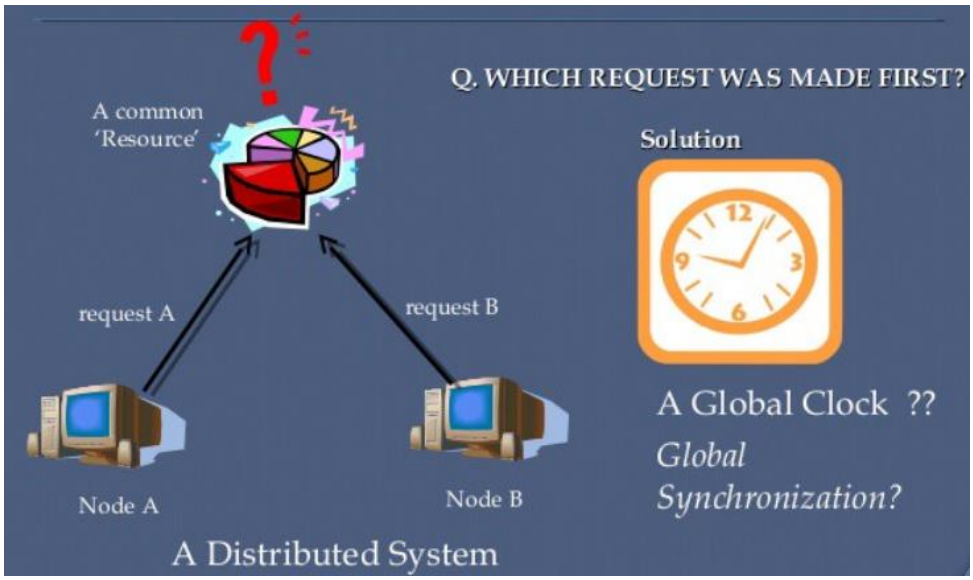
Every computer needs a timer mechanism (called a computer clock)
to keep track of current time and also For various accounting purposes such as calculating the time spent by a process in CPU utilization, disk I/O and so on, so that the corresponding user can be charged properly.

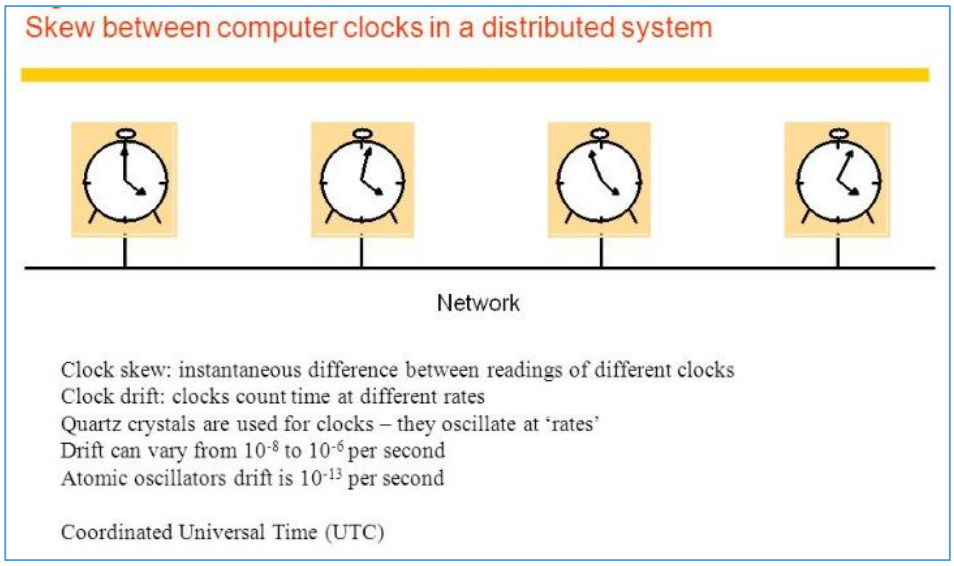
In a distributed system, an application may have processes that concurrently run on multiple nodes of the system. For correct results, several such distributed applications require that the clocks of the nodes are subchronised with each other.

For example, for a distributed-on line reservation system to be fair, the only remaining seat booked almost simultaneously from two different nodes should be offered to the client who booked first, even if the time different between the two bookings is very small. It may not be possible to guarantee this if the clocks of the nodes of the system are not synchronized. In a distributed system, synchronized clocks also enable one to measure the duration of distributed activities that start on one node and terminate on another node, for instance calculating the time taken to transmit a message from one node to another at any arbitrary time. It is difficult to get the correct result in the case if the clocks of the sender and receiver nodes are not synchronized.

There are several other applications of synchronized clocks in distributed systems. Some good examples of such applications may be found in [Liskov 1993].

The discussion above shows that it is the job of a **distributed operating system** designer to devise and use suitable algorithms for properly synchronizing the clocks of a distributed system. This section presents a description of such algorithms. However, for a better understanding of these algorithms, we will first discuss how computer clocks are implemented and what are the main issues in synchronizing the clocks of a distributed system?





6.2 HOW COMPUTER CLOCKS ARE IMPLEMENTED

A computer clock usually consists of three components –

- a quartz crystal that oscillates at a well-defined frequency,
- a counter register, and

- a holding register.

The holding register is used to store a constant value that is decided based on the frequency of oscillation 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 holding register. Each interrupt is called clock tick. The value in the holding register is chosen 60 so that on 60 clock ticks occur in a second.

CMOS RAM is also present in most of the machines which keeps the clock of the machine up-to-date even when the machine is switched off. When we consider one machine and one clock, then slight delay in the clock ticks over the period of time does not matter, but when we consider n computers having n crystals, all running at a slightly different time, then the clocks get out of sync over the period of time. This difference in time values is called clock skew.

Tool Ordering of Events:

We have seen how a system of clocks satisfying the clock condition can be used to order the events of a system based on the happened before relationship among the events. We simply need to order the events by the times at which they occur. However that the happened before relation is only a partial ordering on the set of all events in the system. With this event ordering scheme, it is possible that two events a and b that are not related by the happened before relation (either directly or indirectly) may have the same timestamps associated with them. For instance, if events a and b happen respectively in processes $P1$ and $P2$, when the clocks of both processes show exactly the same time (Say 100), both events will have a timestamp of 100. In this situation, nothing can be said about the order of the two events. Therefore, for total ordering on the set of all system events an additional requirement is desirable. No two events ever occur all exactly the same time. To fulfill this requirement, Lamport proposed the use of any arbitrary total ordering of the processes. For example, process identity numbers may be used to break ties and to create a total ordering on events. For instance, in the situation described above, the timestamps associated with events a and b will be 100.001 and 100.002, respectively, where the process identity numbers of processes $P1$ and $P2$ are 001 and 002 respectively. Using this method, we now have a way to assign a unique timestamp to each event in a distributed system to provide a total ordering of all events in the system.

6.3 MUTUAL EXCLUSION

There are several resources in a system that must not be used simultaneously by multiple processes if program operation is to be correct. For example, a file must not be simultaneously updated by multiple processes. Similarly, use of unit record peripherals such

as tape drives or printers must be restricted to a single process at a time. Therefore, exclusive access to such a shared resource by a process must be ensured. This exclusiveness of access is called mutual exclusion between processes. The sections of a program that need exclusive access to shared resources are referred to as critical sections. For mutual exclusion, means are introduced to prevent processes from executing concurrently within their associated critical sections.

An algorithm for implementing mutual exclusion must satisfy the following requirements:

Issues in Recovery from Deadlock:

Two important issues in the recovery action are **selection of victims** and **use of transaction mechanism**. These are described below.

Selection of Victim(s): In any of the recovery approaches described above, deadlock is broken by killing or rolling back one or more processes. These processes are called victims. Notice that even in the operator intervention approach, recovery involves killing one or more victims. Therefore, an important issue in any recovery procedure is to select the victims.
Selection of victim(s) is normally based on two major factors:

1. **Minimization of recovery cost:** This factor suggests that those processes should be selected as victims whose termination / rollback will incur the minimum recovery cost. Unfortunately, it is not possible to have a universal cost function, and therefore, each system should determine its own cost function to select victims.
   Some of the factors that may be considered for this purpose are
   a) the priority of the processes;
   b) the nature of the processes, such as interactive or batch and possibility of return with no ill effects;
   c) the number and types of resources held by the processes;
   d) the length of service already received and the expected length of service further needed by the processes; and
   e) the total number of processes that will be affected.

2. **Prevention of starvation:** If a system only aims at minimization of recovery cost, it may happen that the same process (probably because its priority is very low) is repeatedly selected as a victim and may never complete. This situation known as starvation, must be somehow prevented in any practical system. One approach to handle this problem is to raise the priority of the process every time it is victimized. Another approach is to include the number of times a process is victimized as a parameter in the cost function.

Use of Transaction Mechanism: After a process is killed or rolled back for recovery from deadlock, it has to be returned. However, rerunning a process may not always be safe, especially when the operations already performed by the process are non-idempotent. For example, if a process has updated the amount of a bank account by adding a certain amount to it, re-execution of the process will result in adding the same amount once again, leaving the balance in the account in an incorrect state. Therefore, the use of transaction mechanism (which ensures all or no effect) becomes almost inevitable for most processes when the system chooses the method of detection and recovery for handling deadlocks. However, notice that the transaction mechanism need not be used for those processes that can be rerun with no ill effects. For example, rerun of a compilation process has no ill effects because all it does is read a source file and produce an object file.

6.4 ELECTION ALGORITHMS

Several distributed algorithms require that there be a coordinator process in the entire system that performs some type of coordination activity needed for the smooth running of other processes in the system. Two examples of such coordinator processes encountered in this chapter are

- the coordinator in the centralized algorithm for mutual exclusion and
- the central coordinator in the centralized deadlock detection algorithm.

Since all other processes in the system have to interact with the coordinator, they all must unanimously agree on who the coordinator is. Furthermore, if the coordinator process fails due to the failure of the site on which it is located, a new coordinator process must be elected to take up the of the failed coordinator.

Election algorithms are meant for electing to take coordinator process from among the currently running processes in such a manner that at any instance of time there is a single coordinator for all processes in the system.

Election algorithm are based on the following assumptions :

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

Therefore, whenever initiated, an election algorithm basically finds out which of the currently active processes has the highest priority number and then informs this to all other active processes. Different election algorithms differ in the way they do this. Two such election algorithms are described below.

6.4.1 Bully Algorithm:

When any process notices that the coordinator is no longer responding to the requests, it asks for the election.

Example: A process P holds an election as follows

1) P sends an ELECTION message to all the processes with higher numbers.

2) If no one responds, P wins the election and becomes the coordinator.

3) If one higher process answers; it takes over the job and P's job is done.

At any moment an "election" message can arrive to process from one of its lowered numbered colleague. The receiving process replies with an OK to say that it is alive and can take over as a coordinator. Now this receiver holds an election and in the end all the processes give uo except one and that one is the new coordinator.

The new coordinator announces its new post by sending all the processes a message that it is starting immediately and is the new coordinator of the system.

If the old coordinator was down and if it gets up again; it holds for an election which works in the above mentioned fashion. The biggest numbered process always wins and hence the name "bully" is used for this algorithm.

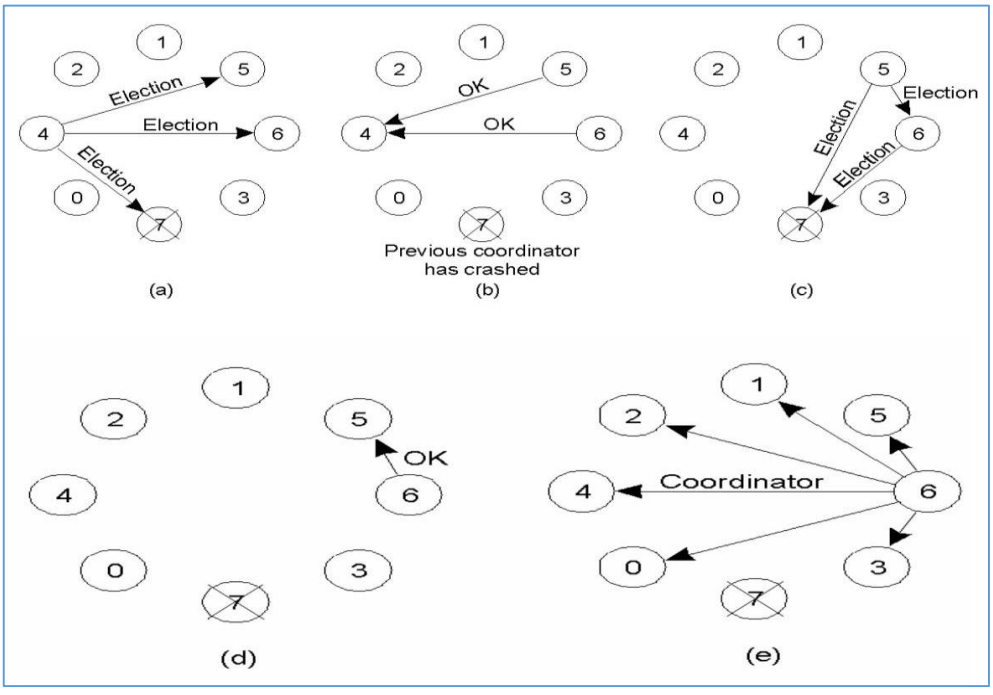


Figure 6.1: The bully election algorithm

a) Process 4 holds an election.

b) Process 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.

6.4.2 Ring Algorithm:

It is based on the use of a ring as the name suggests. But this does not use a token. Processes are physically ordered in such a way that every process knows its successor.

When any process notices that the coordinator is no longer functioning, it builds up an ELECTION message containing its own number and passes it along the to its successor. If the successor is down, then sender skips that member along the ring to the next working process.

At each step, the sender adds its own process number to the list in the message effectively making itself a candidate to be elected as the coordinator. At the end, the message gets back to the process that started it.

That process identifies this event when it receives an incoming message containing its own process number. Then the same message is changed as coordinator and is circulated once again.

Example: two process, Number 2 and Number 5 discover together that the previous coordinator; Number 7 has crashed. Number 2 and Number 5 will each build an election message and start circulating it along the ring. Both the messages in the end will go to Number 2 and Number 5 and they will convert the message into the coordinator with exactly the same number of members and in the same order. When both such messages have gone around the ring, they both will be discarded and the process of election will re-start.

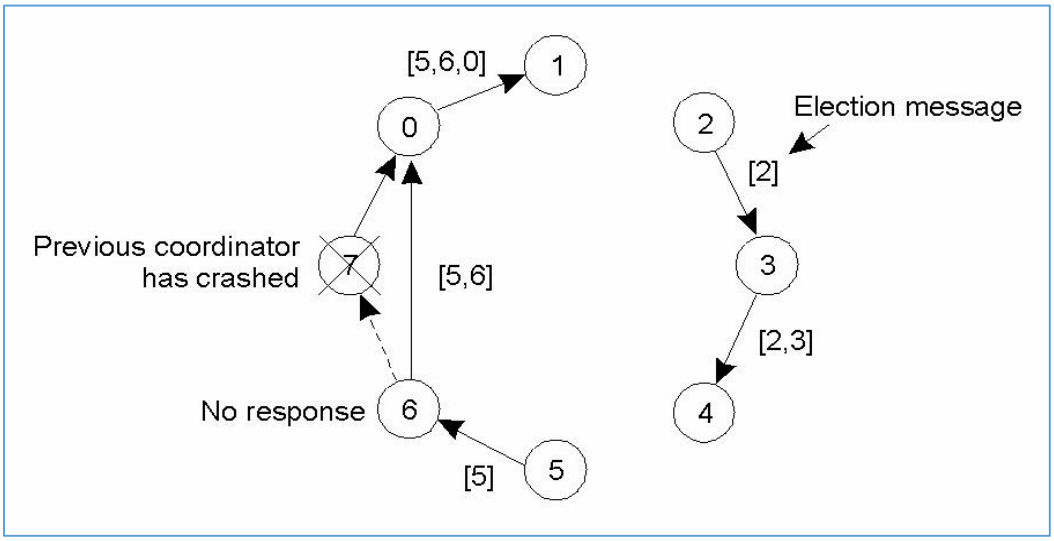


Figure6.2: Election algorithm using Ring

6.5 MUTUAL EXCLUSION

Mutual exclusion (often abbreviated to mutex) algorithms are used in concurrent programming to avoid the simultaneous use of a common resource, such as a global variable, by pieces of computer code called critical sections. A critical section is a piece of code in which a process or thread accesses a common resource. The critical section by itself is not a mechanism or algorithm for mutual exclusion. A program, process, or thread can have the critical section in it without any mechanism or algorithm which implements mutual exclusion.

Examples of such resources are fine-grained flags, counters or queues, used to communicate between code that runs concurrently, such as an application and its interrupt handlers. The synchronization of access to those resources is an acute problem because a thread can be stopped or started at any time. A mutex is also a common name for a program object that negotiates mutual exclusion among threads, also called a lock.

Following are the algorithms for mutual exclusion:

6.5.1 Centralized Algorithm

Here one process is selected as the coordinator of the system with the authority of giving access to other process for entering the critical region. If any process wants to enter the critical, it has to take the permission from the coordinator process. This permission is taking by sending a REQUEST message.

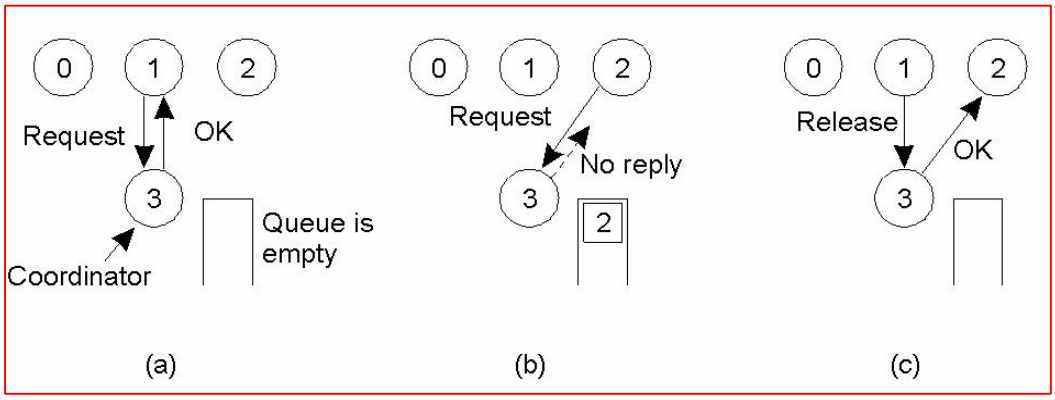


Figure: 6.3:

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, when then replies to process 2.

As shown in figure 5.3-a), the coordinator is not reply to process 2 when the critical region is occupied. Here, depending on the type of system, the coordinator can also reply back to the process 2 that it is in queue. If the coordinator doesn't do so, then the waiting process 2 will be unable to distinguish between 'permission denied" or a "dead" coordinator.

This type of system as a single point of failure, if the coordinator fails, then the entire system crashes.

6.5.2 Distributed Algorithm:

A distributed algorithm for mutual exclusion is presented. No particular assumptions on the network topology are required, except connectivity; the communication graph may be arbitrary. The processes communicate by using messages only and there is no global controller. Furthermore, no process needs to know or learn the global network topology. In that sense, the algorithm is more general than the mutual exclusion algorithms which make use of an a priori knowledge of the network topology.

When a process wants to enter a critical region, it builds a message containing:

- name of the critical region
- it's process number
- It's current time.

The process sends this message to all the processes in the network. When another process receives this message, it takes the action pertaining on its state and the critical region mentioned. Three cases are possible here:

0) If the message receiving process is not in the critical region and does not wish to enter it, it sends it back.
1) Receiver is already in the critical region and does not reply
2) Receiver wants to enter the same critical region and a
3) Has not done so, it compares the "time stamp" of the incoming message with the one it has sent to others for permission. The lowest one wins and can enter the critical region.

When the process exists from the critical region, it sends an OK message to inform everyone.

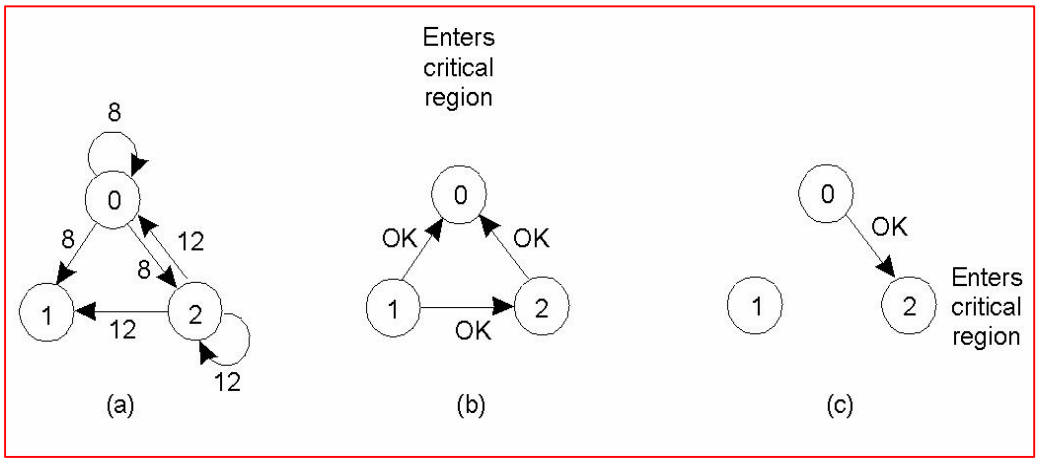


Figure 6.4:

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.**

Disadvantage:

1) If one process crashes, it will fail to respond. Thus other possesses will assume that the process is still working in the critical region which will make other processes go through a starvation.

2) Here each process must maintain the group membership list that includes processes entering or leaving the group.

3) Here all the processes are involved in all decisions; this could lead to bottle neck when the numbers of processes in the group are more.

4) This algorithm is comparatively expensive, slower and complex.

6.5.3 Token Ring Algorithm:

Here we have a bus network (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. 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.

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 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.

The correctness of this algorithm is easy to see. Only one process has the token at any instant, so only one process can actually be in a critical region. Since the token circulates among the processes in a well-defined order, starvation cannot occur. Once a process decides it wants to enter a critical region, at worst it will have to wait for every other process to enter and leave one critical region.

As usual, this algorithm has problems too. If the token is ever lost, it must be regenerated. In fact, detecting that it is lost is difficult, since the amount of time between successive appearances of the token on the network is unbounded. The fact that the token has not been spotted for an hour does not mean that it has been lost; somebody may still be using it.

The algorithm also runs into trouble if a process crashes, but recovery is easier than in the other cases. If we require a process receiving the token to acknowledge receipt, a dead process will be detected when its neighbour tries to give it the token and fails. At that point the dead process can be removed from the group, and the token holder can throw the token over the head of the dead process to the next member down the line, or the one after that, if necessary. Of course, doing so requires that everyone maintains the current ring configuration.

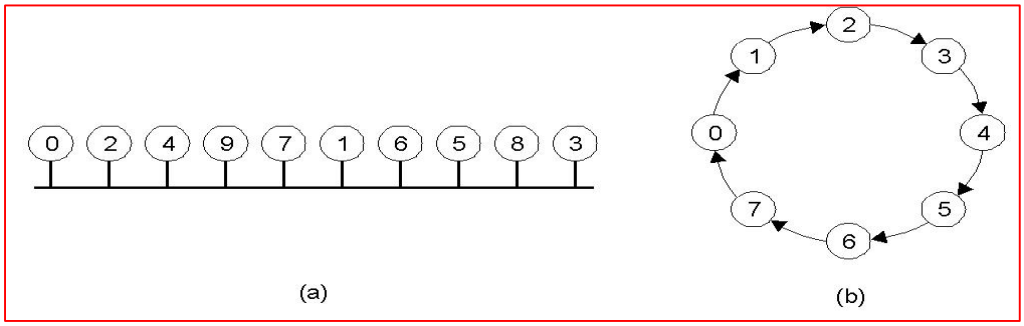


Figure 5.5:

a) An unordered group of processes on a network.

b) A logical ring constructed in software.

Exercises :

6.1 Write pseudocode for an algorithm that decides whether a given set of clocks are synchronized or not. What input parameters are needed in your algorithm?

6.2 How do clock synchronization issues differ in centralized and distributed computing systems?

A resource manager schedules the processes in a distributed system to make use of the system resources in such a manner that resource usage, response time, network congestion, and scheduling overhead are optimized. A varied of widely differing techniques and methodologies for scheduling processes of a distributed system have been proposed. These techniques can be broadly classified into three types :

1. Task assignment approach, in which each process submitted by a user for processing is viewed as a collection of related tasks and these tasks are scheduled to suitable nodes so as to improve performance.

2. Load-balancing approach, in which all the processes submitted by the users are distributed among the nodes of the system so as to equalize the workload among the nodes.

3. Load sharing approach, which simply attempts to conserve the ability of the system to perform work by assuring that no node is idle while processes wait for being processed.

Of the three approaches, the task assignment approach has limited applicability in practical situations because it works on the assumption that the characteristics of all the processes to be scheduled are known in advance. Furthermore, the scheduling algorithms that fall in this category do not normally take care of the dynamically changing date of the system. Therefore, this approach will be covered very briefly just to give an idea of now it works. Before presenting a description of each of these techniques, the desirable features of a good global scheduling algorithm are presented.

Revision Exercise:

1) How do clock synchronization issues differ in centralized and distributed computing systems?

2) What is a dead lock? What are the necessary condition which lead to a dead lock?

3) Explain the Bully's algorithm.

4) Explain the concept of logical clocks and their importance in distributed systems.

5) Prove that an unsafe state is not a dead lock.

6) Explain the ring-based algorithm.

7) What will happen if in a bully algorithm for electing a coordinator when two or more processes almost simultaneously discover that the coordinator has crashed.

8) Why are election algorithms needed is a distributed system?

9) How are false dead locks detected by the dead lock detection systems?

REFERENCES:

Ghosh, Sukumar (2007), Distributed Systems – An Algorithmic Approach, Chapman & Hall/CRC, ISBN 978-1-58488-564-1.

Lynch, Nancy A. *(1996), Distributed Algorithms,* Morgan Kaufmann, ISBN 1-55860-348-4.