# Synchronization

Lecturer: Dr. Joseph Doyle

# Introduction

- Events occur at different nodes in the network
- These events are recorded using different clocks
- How do other nodes in the network determine when events took place

# Why is it important

- May seem like a trivial issue but actually very important
- How can we determine if data is up-to-date unless we can determine the order of writes to the data
- Is it possible to sell stock that does not exist?

# Synchronize Clocks

- Simplest solution would be to synchronise clocks on every node in the network
- In practice this is very difficult to achieve

# Hardware Clocks

- Hardware clocks in computers are crystal oscillators which are connected to oscillation counter circuitry
- This counter is then scaled to provide an approximate representation of physical time

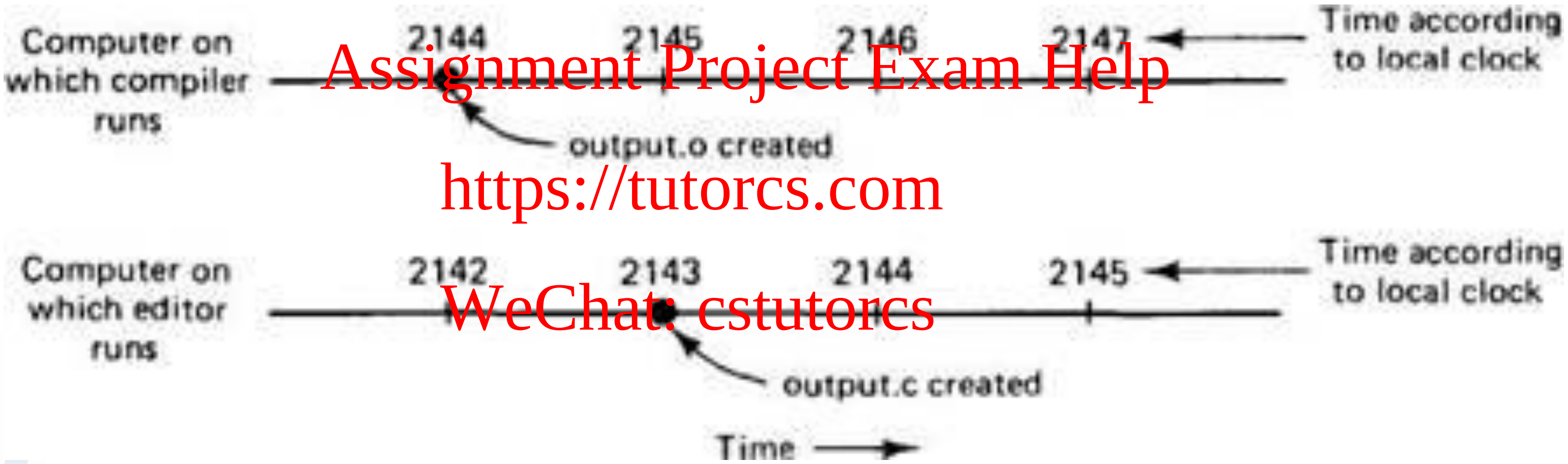
# Problems with Hardware Clocks

- Clocks are not all exactly the same
- The clock rate is affected by
  - Physical differences in the crystal
  - Temperature differences
  - Voltage
  - Humidity

# Problems with Hardware Clocks

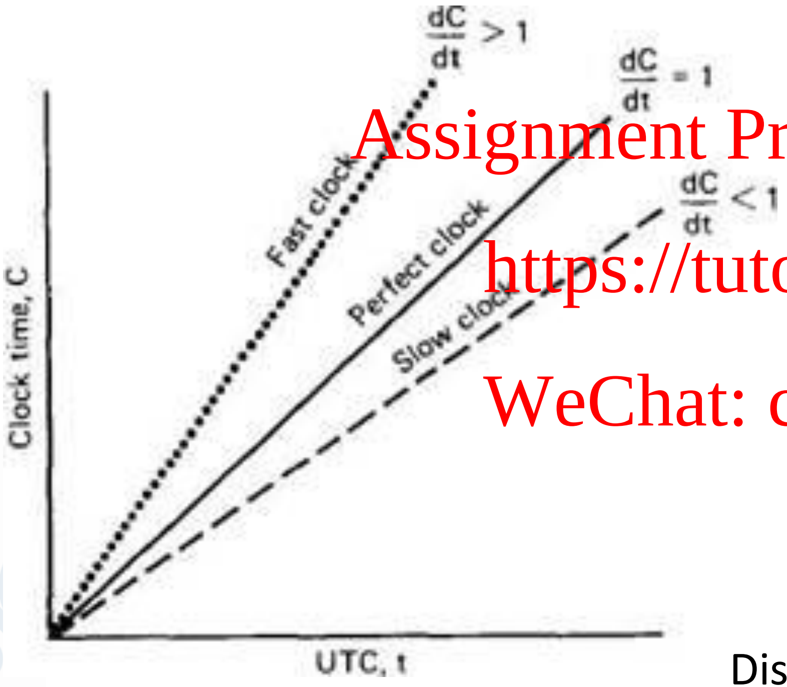
- This results in three problems:
  - **Skew:** This is a disagreement in the reading of two clocks
  - **Drift:** This is the difference in the rate at which two clocks count the time
  - **Clock Drift Rate:** This is the difference in precision between a hardware clock and a perfect reference clock. Approximately $10^{-6}$sec/sec in normal clocks and $10^{-7}$ - $10^{-8}$ sec/sec in high precision clocks

# Clock Synchronization


Computer on which compiler runs
2144
2145
2146
2147
Time according to local clock
output.o created
Computer on which editor runs
2142
2143
2144
2145
Time according to local clock
output.c created
Time

Distributed Operating Systems. Andrew Tanenbaum

# Clock Synchronization



- Assuming a normal clock drift rate of $10^{-6}$ sec/sec a drift of 1ms occurs in approximately 17 minutes and a drift of 1s occurs in approximately 12 days

Distributed Operating Systems. Andrew Tanenbaum

# Synchronizing Clocks

- External Synchronization
  - An external authoritative clock updates the clock of nodes in the network
  - Skew is limited to the delay between the authoritative clock server and the node
- Internal Synchronization
  - Nodes in the network collaborate to limit the skew to a delay bound
  - Delay is larger here as a round trip is required

# Software Based Clock Synchronization

- Christian’s Algorithm
- Berkley Algorithm
- Network Time Protocol (Internet)

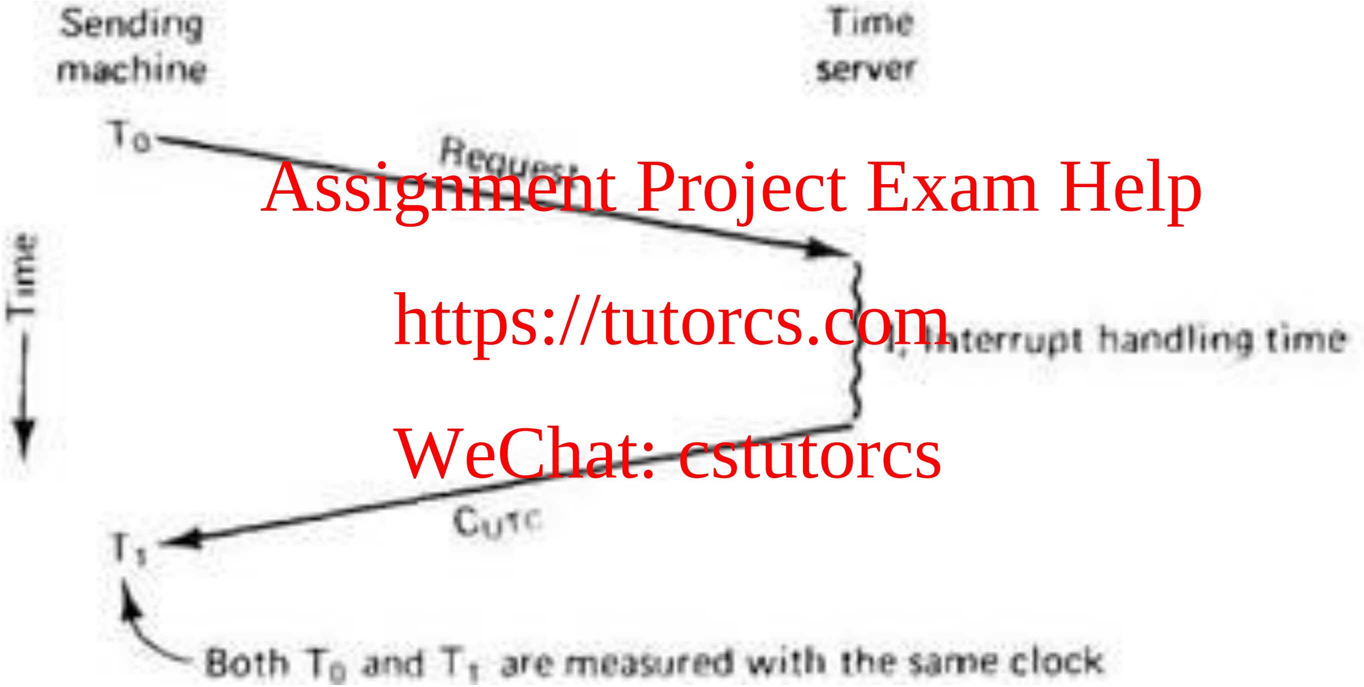
# Christian's Algorithm

- Based around the observation that round trip time in LAN networks is sufficiently small to ensure reasonable clock accuracy
- It requires a clock server which is synchronized to UTC time

# Christian’s Algorithm

- A node in the network sends a request to the clock server and records the round trip time RTT
- The clock server measures its current time T and sends this to the node
- The node then updates its time t to be t=T+RTT/2

# Christian’s Algorithm


Sending machine
Time server
$T_0$
Request
Time
I, Interrupt handling time
$C_{UTC}$
$T_1$
Both $T_0$ and $T_1$ are measured with the same clock

Distributed Operating Systems. Andrew Tanenbaum

# Christian’s Algorithm Problems

- Assumes that the duration of the two parts of the round trip are equal
- Not really suitable outside a LAN as the RTT can increase dramatically
- The clock server is a central point of failure

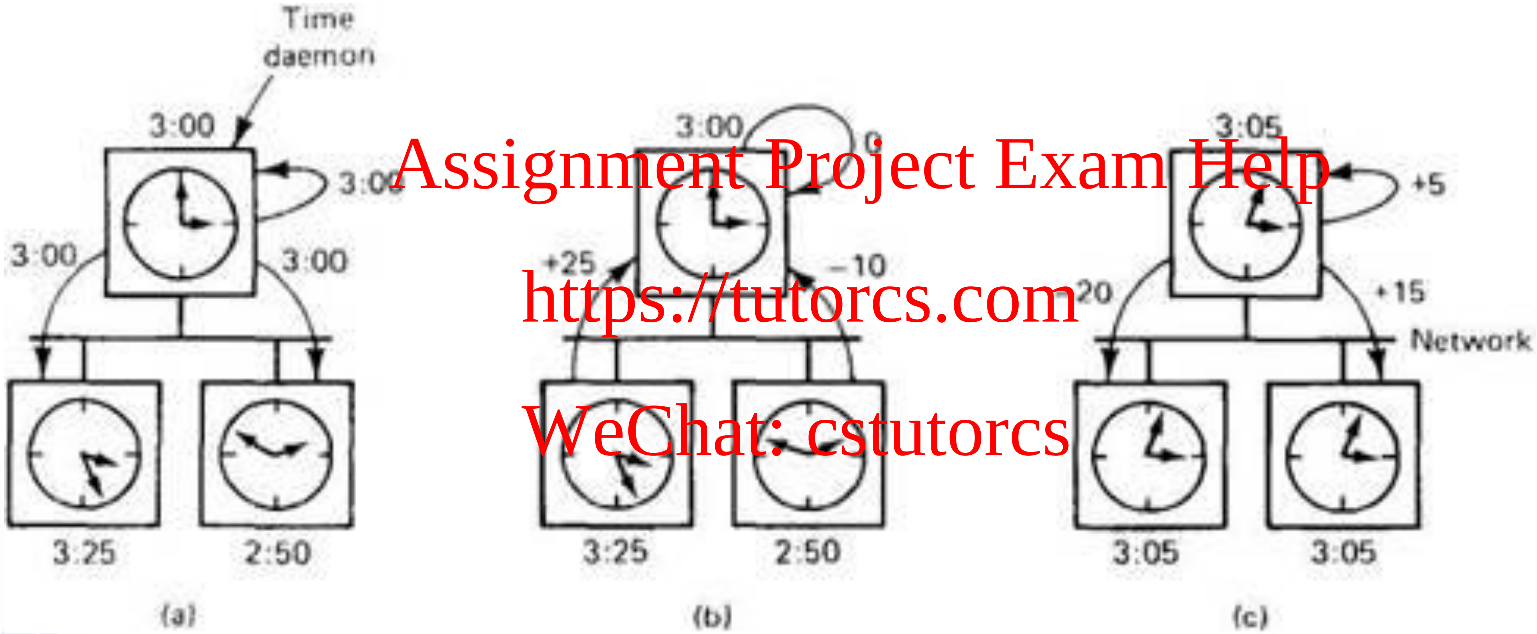
# Berkeley Algorithm

- Variation of Christians Algorithm which does not require a clock server that is synchronized to UTC time
- Uses a manager server instead of a clock server to alter the clocks of nodes in the network

# Berkeley Algorithm

- Manager server periodically polls all nodes in the network
- It records the RTT and uses this to estimate the clock of the node in a similar fashion to Christian's algorithm
- It averages the values obtained from all nodes
- It instructs the nodes to alter their clocks based upon this average to synchronize the times of the nodes
- If the manager fails a new manager is elected using a manager election algorithm (This is discussed later)

# Berkeley Algorithm



Distributed Operating Systems. Andrew Tanenbaum

# Berkeley Algorithm Problems

- Again this is really only suitable for LAN networks
- Also while the nodes are synchronized with each other they are not synchronized with external systems if this is required as is the case with Christian’s algorithm

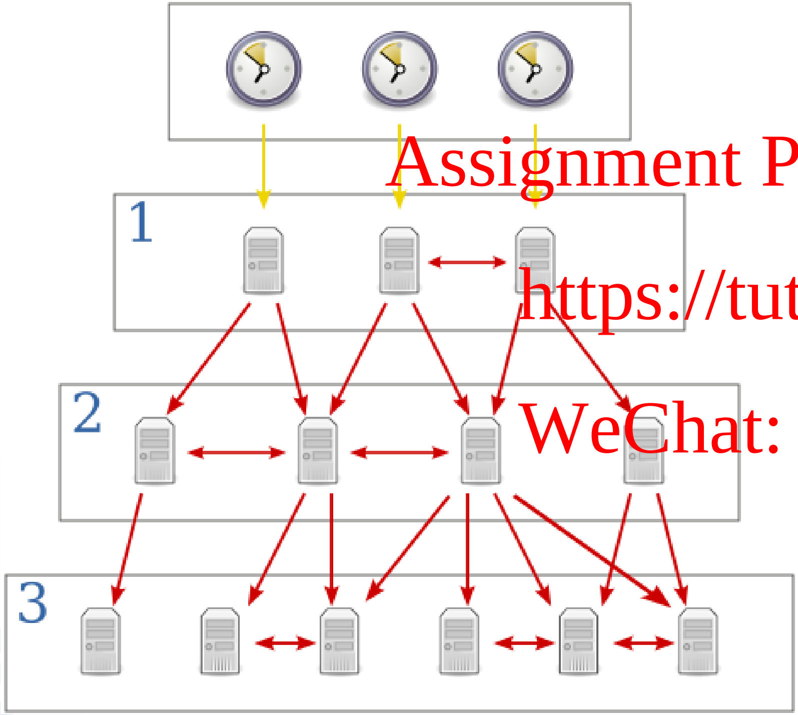
# Network Time Protocol (NTP)

- The goal of this protocol is to improve on the previously discussed algorithms to allow:
  - The ability to synchronize clients via the Internet
  - Reliable service in the event of lengthy losses of connectivity
  - Provide protection against interference

# Network Time Protocol (NTP)

- NTP uses a hierarchical model with different stratum in the hierarchy assigned different levels of accuracy
  - **Stratum 0** are high precision timekeeping devices such as atomic clocks, GPS or other radio clocks. They are know as reference clocks
  - **Stratum 1** are computers which are synchronised to with a few microseconds of a Stratum 0 devices. These are the primary time servers.
  - **Stratum 2** are nodes connected to primary time servers. It is possible to connect to multiple primary time servers

# Network Time Protocol (NTP)



https://en.wikipedia.org/wiki/Network_Time_Protocol

# Network Time Protocol (NTP)

- Up to 15 stratum are possible in NTP
- The higher the stratum number the less accurate the clock of the node due to the increased RTT

# NTP Timestamps

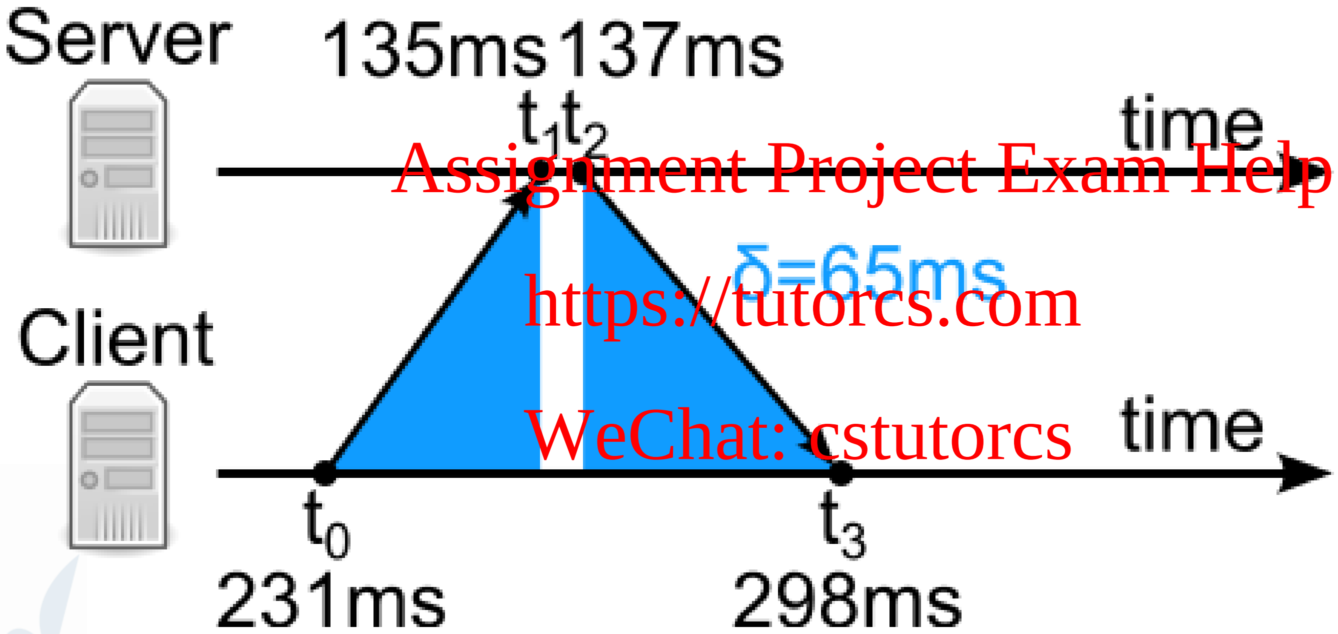
- Uses a 64 bit timestamp with 32 bits for seconds and 32 bits for fractional seconds
- It has a theoretical resolution of 233 picoseconds
- It has a timescale that rolls over ever 136 years. First rollover will occur on February 7th 2036 (Something to look forward to)

# NTP Algorithm

- The NTP client will regularly poll one or more server
- Using this information they will calculate the time offset $\theta$ and round trip delay $\delta$ as:

- $\theta = \frac{(t_1 - t_0)+(t_2 - t_3)}{2}$
- $\delta = (t_3 - t_0) - (t_2 - t_1)$

# NTP Algorithm



https://en.wikipedia.org/wiki/Network_Time_Protocol

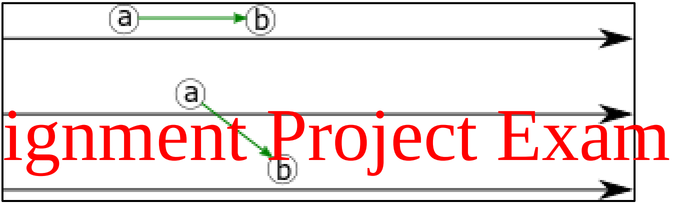
# NTP Algorithm

- Once these values are returned to the client they are passed through a statistical filter to eliminate outliers
- An estimate of the offset is calculated from the best three remaining candidates (It favours messages from higher stratum)
- The client’s clock is then adjusted gradually to reduce the offset

# Synchronization Requirements

- **Causality:** real-time order ~ timestamp order
- **Groups/replicated:** All members of the group see events in the same order
- **Multiple-copy-updates:** order of the updates, consistency conflicts
- **Serializability of transactions:** Common order of transaction order

# Happened-Before Relations (a->b)



- a and b are defined as events in the same process
- If a occurs before b then a->b
- For example if a is a message being sent and b is a message being received then a->b
- a| | b if neither a->b and b->a (a and b are concurrent)
- If a->b and b->c then a->c

# Lamport Timestamps

- The rules for this algorithm are as follows:
  - A process increments its counter before each event that it processes
  - The process includes this counter when it sends a message
  - On receiving a message the counter of the recipient is updated if necessary to the greater of the its current counter and timestamp received in its message. The counter is then incremented by one to indicate that the message has been received.

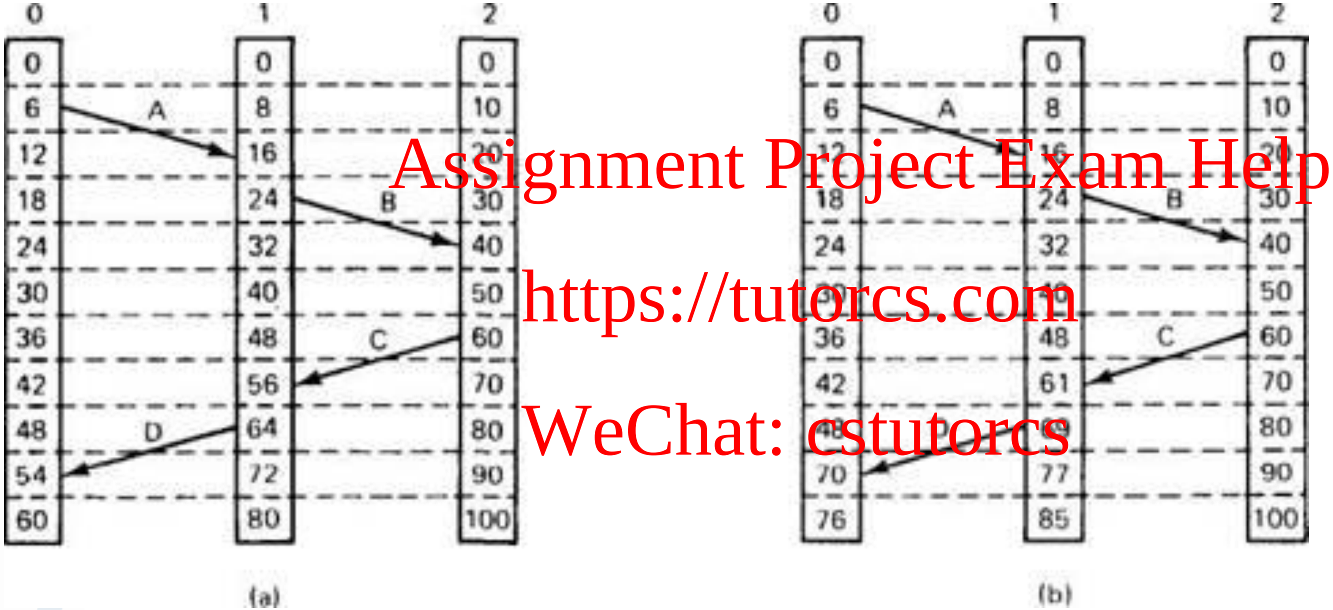
# Lamport Timestamps

- If we define a process as $\boldsymbol{p_i}$, an event as $\boldsymbol{e}$, a counter as $\boldsymbol{L_i}$ and a timestamp as $L_i(e)$ we can define this algorithm as

- At $p_i$: before each event $L_i = L_i + 1$

- When $p_i$ sends a **message m** to $p_j$
  - $p_i: L_i = L_i + 1$; t= $L_i$; $message$ = (m,t);
  - $p_j$: $L_j = max(L_j, \mathrm{t})$; $L_j = L_j + 1$
  - $L_j(receive\ event) = L_j$

# Lamport Timestamps



(Uncorrected)

(Lamport Timestamps)

Distributed Operating Systems. Andrew Tanenbaum

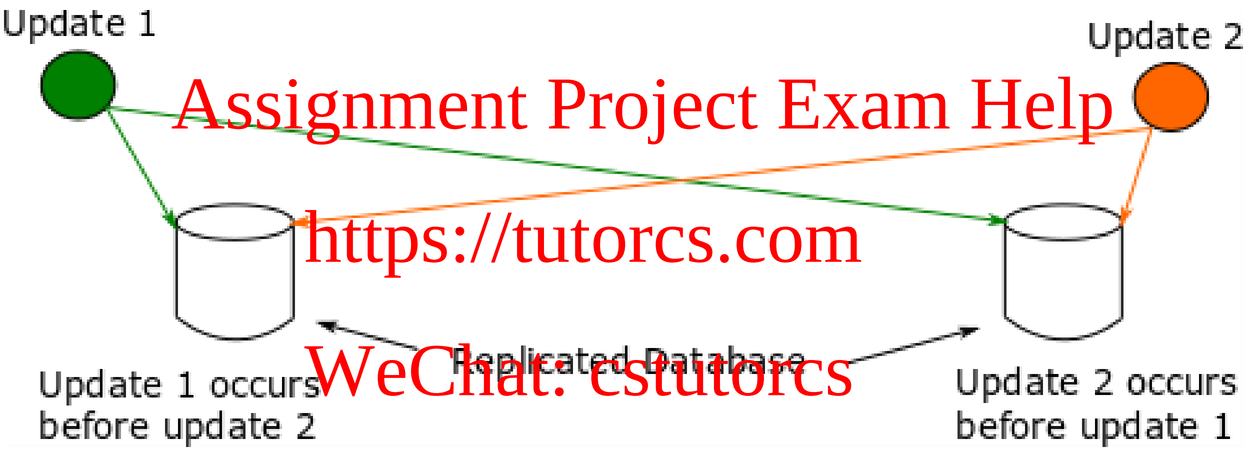
# Lamport Timestamps Problems

- Suppose there are two event a and b
- If there is any way that event a could have influenced event b then we can state that a->b and the Lamport timestamp of event a is less than the Lamport timestamp of event b
- However, if we have two Lamport timestamps L(a) < L(b) we cannot explicitly state that a->b as Lamport timestamps do not fulfil the **strong clock consistency condition**
- Lamport timestamps only create a **partial ordering** of events
- They can be used to create a **total ordering** of events by including an arbitrary mechanism to break ties (albeit with the caveat that this cannot be used to imply a causal relationship)

# Total Ordered Lamport Timestamps

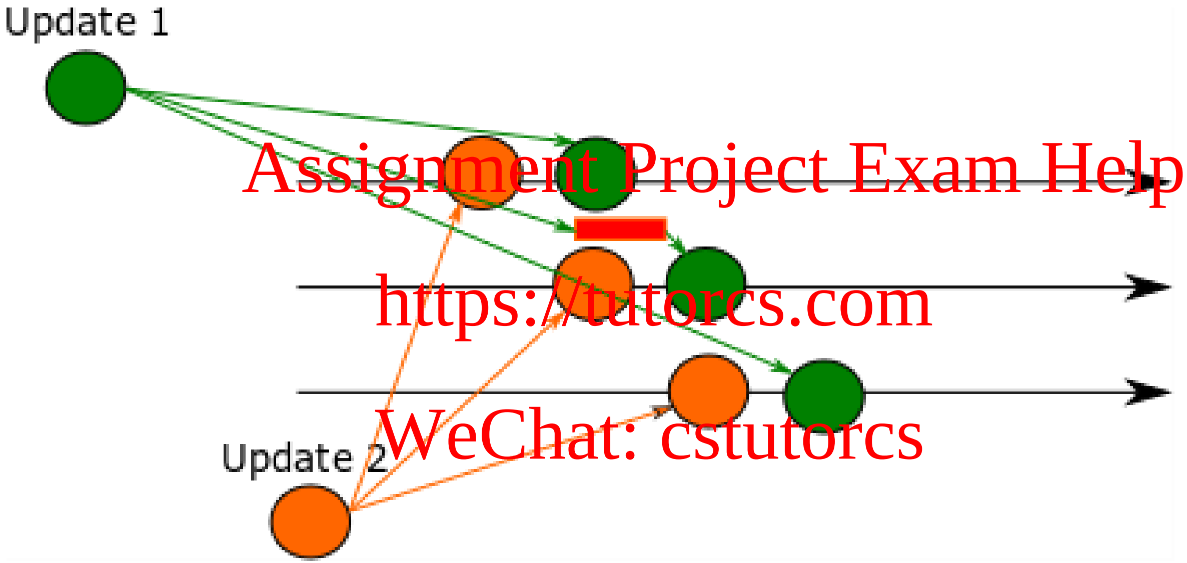
- Expand Timestamp to $(\boldsymbol{L_i}(\boldsymbol{e}), \boldsymbol{i})$
- $(L_i(e), i) < (L_j(e), j)$
- If $L_i(e) < L_j(e)$ or ($L_i(e) = L_j(e)$ and $i < j$)

# Total Order Multicasting Examples


Update 1
Update 2
Replicated Database
Update 1 occurs before update 2
Update 2 occurs before update 1

Database is updated leaving an inconsistent state

# Total Order Multicasting Examples



All receivers see all messages in the same order
(which is not necessarily the original sending order)
E.g. Group Updates

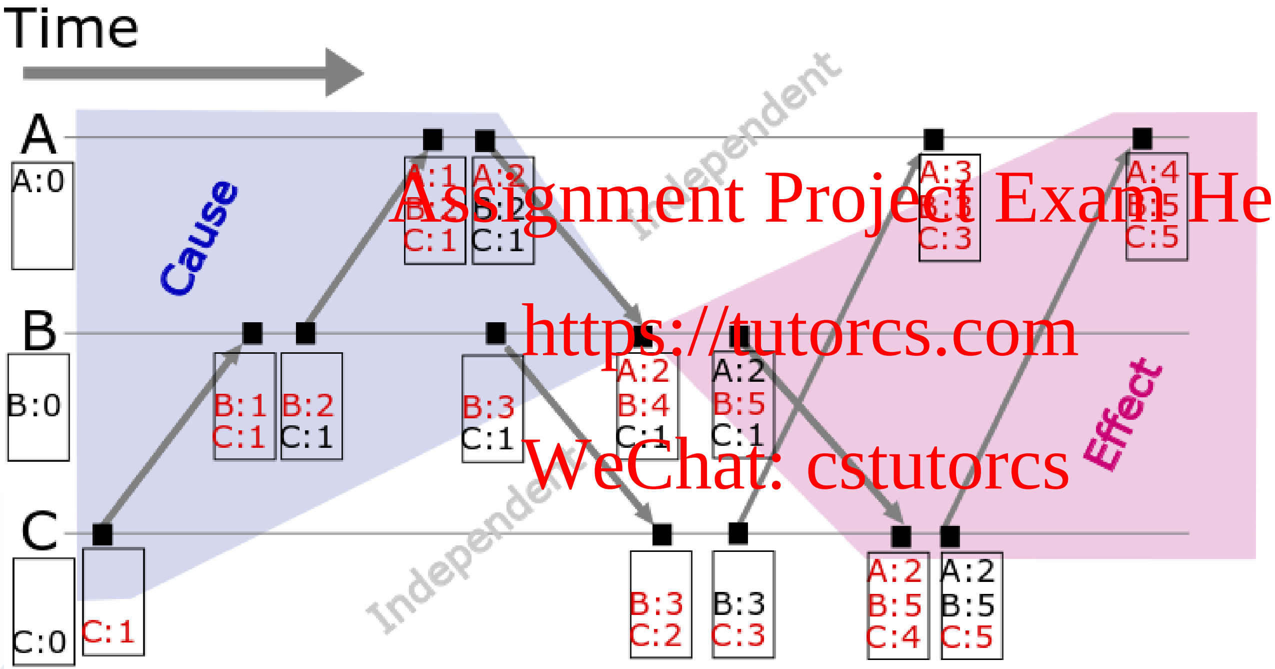
# Vector Clocks

- Vector clocks expand on Lamport Timestamps by including additional data for detecting causality violations
- Each Process $P_i$ maintains a vector $V_i$
- $V_i[i]$ is the number of events that have occurred at $P_i$
- If $V_i[j] = k$ is then $P_i$ knows about the $k$ events that have occurred at $P_j$

# Vector Clocks

- Order of timestamps
- $V = V'$ iff $V[j] = V'[j]$ for all j
- $V \leq V'$ iff $V[j] \leq V'[j]$ for all j
- $V < V'$ iff $V \leq V'$ and $V \neq V'$
- Order of events
- $e \rightarrow e'$ => $V(e) < V(e')$
- $V(e) < V(e')$ => $e \rightarrow e'$
- $e \,||\, e'$ if not $V(e) \leq V(e')$
- and not $V(e') \leq V(e)$

# Vector Clocks



https://en.wikipedia.org/wiki/Vector_clock

# Vector Clocks

- The algorithm for vector clocks is as follows:
- $P_i$ multicast $V_i[i]$ = $V_i[i]$ + 1
- Each message includes $V_i$
- For each $P_j$ which is receiving a message: The message can be delivered when
  - $V_i[i]$ = $V_j[i]$ + 1 (All previous message from *i* have arrived)
  - $V_j[k]$ >= $V_i[k]$ for all k, k≠i (*j* has seen all the message *i* has seen when the message was sent)
- Upon delivery of message $V_j[i]$ = $V_j[i]$ + 1

# Global State

- Timestamps can be used for a variety of purposes such saving a snapshot of a distributed system

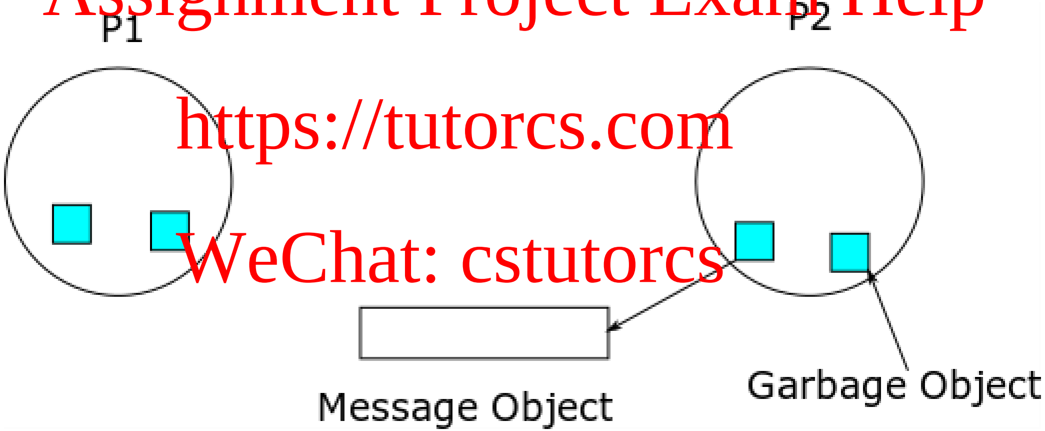
- To create a snapshot the system needs information on:
  - The state of processes
  - Messages in transfer

- There are potential problems which can affect this snapshot namely:
  - Garbage Collection
  - Deadlock
  - Termination

# Garbage Collection

- Garbage collection is an automatic form of memory management
- Very important on distributed systems as manual memory management is more difficult when using multiple systems
- For example consider a simple distributed messaging system
- Messages must be kept until they can be processed after which they can be discarded by a garbage collector
- Need to consider timestamps of messages to determine if they should be included in a snapshot or if they should be discarded

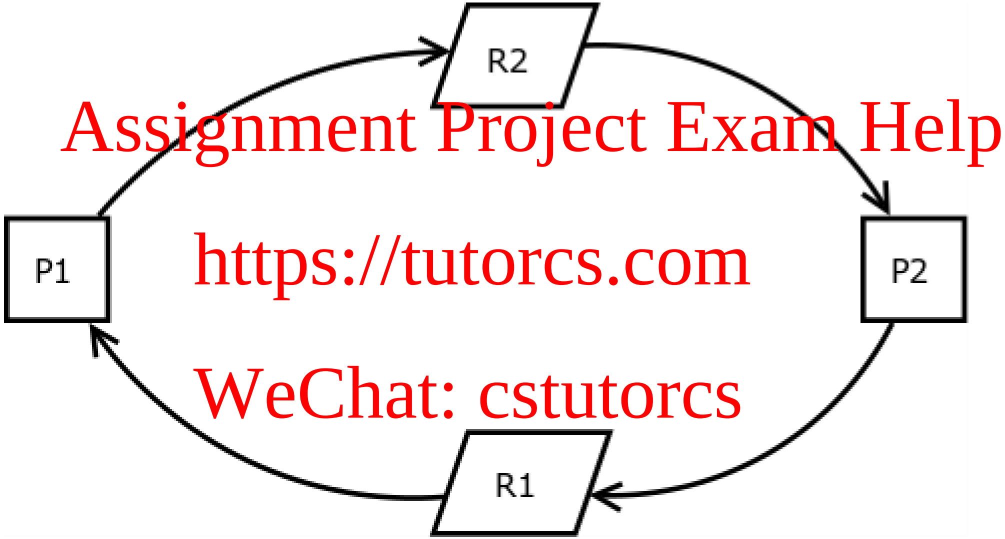
# Garbage Collection



# Deadlock

- Deadlock occurs when two processes are waiting for each other to take an action such as releasing a lock which is associated with a resource
- This can occur for a variety of reasons. Consider the following example
- $P_1$ has a lock on resource $R_1$ and $P_2$ has a lock on resource $R_2$. $P_1$ needs to obtain a lock on $R_2$ before it will release $R_1$ and $P_2$ needs to obtain a lock on $R_1$ before it will release $R_2$.
- Unless action is taken to resolve the deadlock $P_1$ and $P_2$ will continue to wait for a lock they cannot obtain.
- It would be useful to resolve deadlocks before taking a snapshot as they prevent progress in distributed systems

# Deadlock

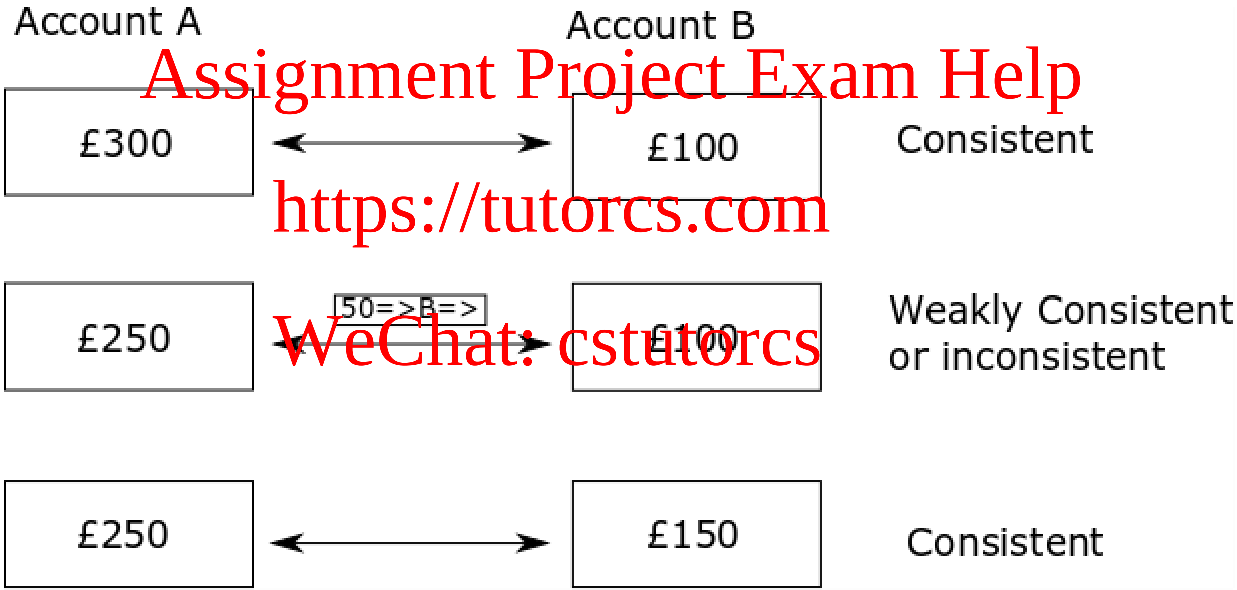


https://en.wikipedia.org/wiki/Deadlock

# Global Snapshot

- To create a global snapshot the following algorithm can be used:
- At each node $i$
  - A local clock records the time $T_i$
  - A state $S_i$ is recorded as a list of tuples containing {event, timestamp}
- The system state S is defined as a vector of the state of each node $S_i$
- A snapshot will contain information on each node up to time *T*
- A snapshot can be considered consistent or inconsistent

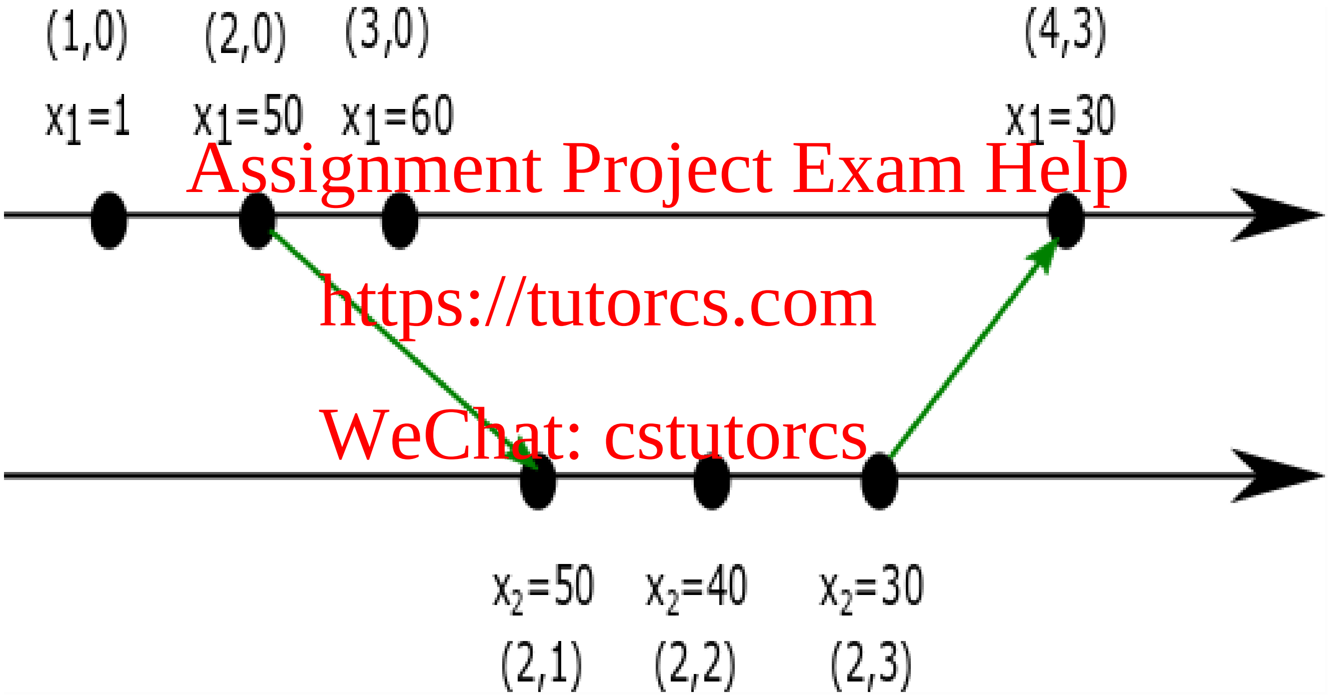
# Inconsistent Snapshot Example



# Consistent Snapshot

- A snapshot is consistent if it contains all the event which happened before the snapshot time *T*
- Consider the example where changes to a local value $x_i$ at processor *i* only need to be sent to another processor when they exceed a certain threshold (>19)
- A vector clock is incremented when $x_i$ changes
- *S* records all changes to the $x_i$ value

# Consistent Snapshot



# Chandy Lamport Algorithm

- To create a consistent global state of a distributed system the Chandy-Lamport Algorithm can be used
- The algorithm assumes that:
  - There are no network failures and all messages arrive intact
  - The snapshot algorithm does not interfere with the normal operation of the processes
- It would be possible to modify this algorithm with TCP/IP to relax the no network failures assumption

# Chandy Lamport Algorithm

- The process which is initiating the snapshot process:
  - Saves its own state
  - Sends a snapshot request to all processes bearing a snapshot token
- When a process receives a snapshot request it
  - Sends the snapshot process its saved state
  - Attaches the snapshot token to all subsequent messages
- When a process receives a message that does not have the snapshot token when it has received the snapshot token it forwards the message to the snapshotting process so that it can be included in the snapshot

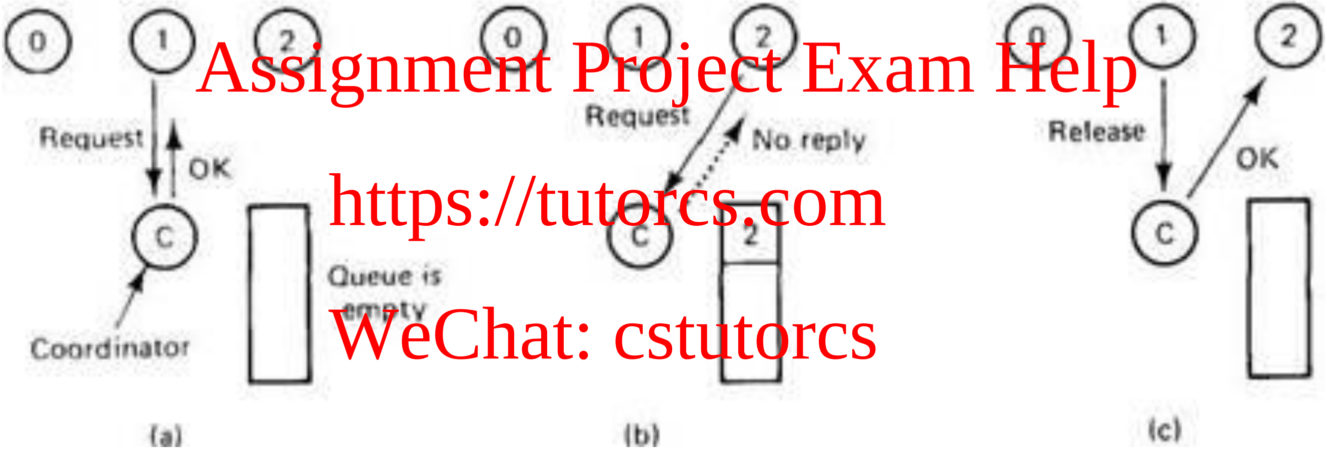
# Resource Reservation

- Timestamps are also useful for reserving resources
- The process which controls resource allocation can be centralized or distributed
- Centralized in general is easy to implement but are not fault tolerant
- Distributed processes more complicated but are in general more fault tolerant
- The other disadvantage of distributed solution is that it tends to be more difficult to debug
- This is sometimes referred to as Mutual Exclusion

# Centralized Resource Allocation

- In the central case a process will ask the coordinator for access to the resource
- The coordinator maintains a queue and if the queue is empty it will grant access to the process
- If another process contacts the coordinator it is placed in the queue and the coordinator does not respond to the request
- When the first process finishes it informs the coordinator and the coordinator then allows the next process in the queue access to the resource

# Centralized Resource Allocation



Distributed Operating Systems. Andrew Tanenbaum

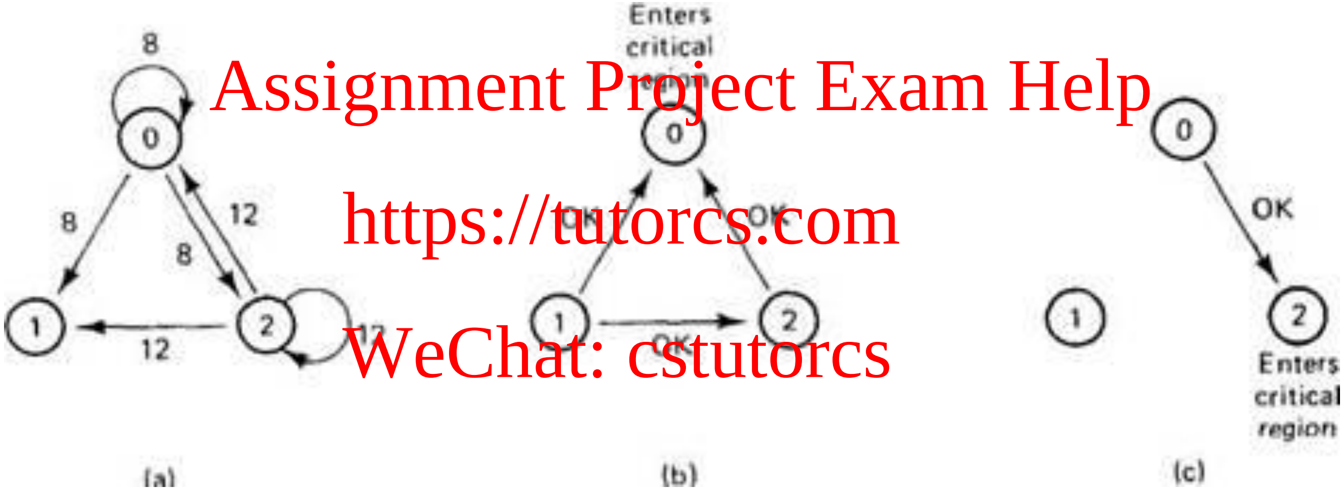
# Centralized Resource Allocation

- The general requirements of mutual exclusion are:
- **Safety:** Only one process is allowed access to the resource at any given time (The algorithm fulfils this requirement)
- **Liveness:** All requests eventually succeed. Deadlock is prevented. (This requirement is not specifically fulfilled. Several methods can be used to break deadlock with the simplest being timeout functionality)
- **Fairness:** If request A happens before request B then A is honoured before B (Timestamps play a role in this requirement at request A may have been delayed in the network but it should still be honoured before request B)

# Distributed Resource Allocation

- Also possible to use a distributed scheme
- When a process wants a resource it multicasts a request to all process and waits for the response
- When a process receives a request if it does not want the response it responds immediately
- If it is using the resource or wants to use the resource and the timestamp of its request is lower than the received timestamp it puts the request in a queue
- When it finishes using the request it replies to all requests in the queue

# Distributed Resource Allocation

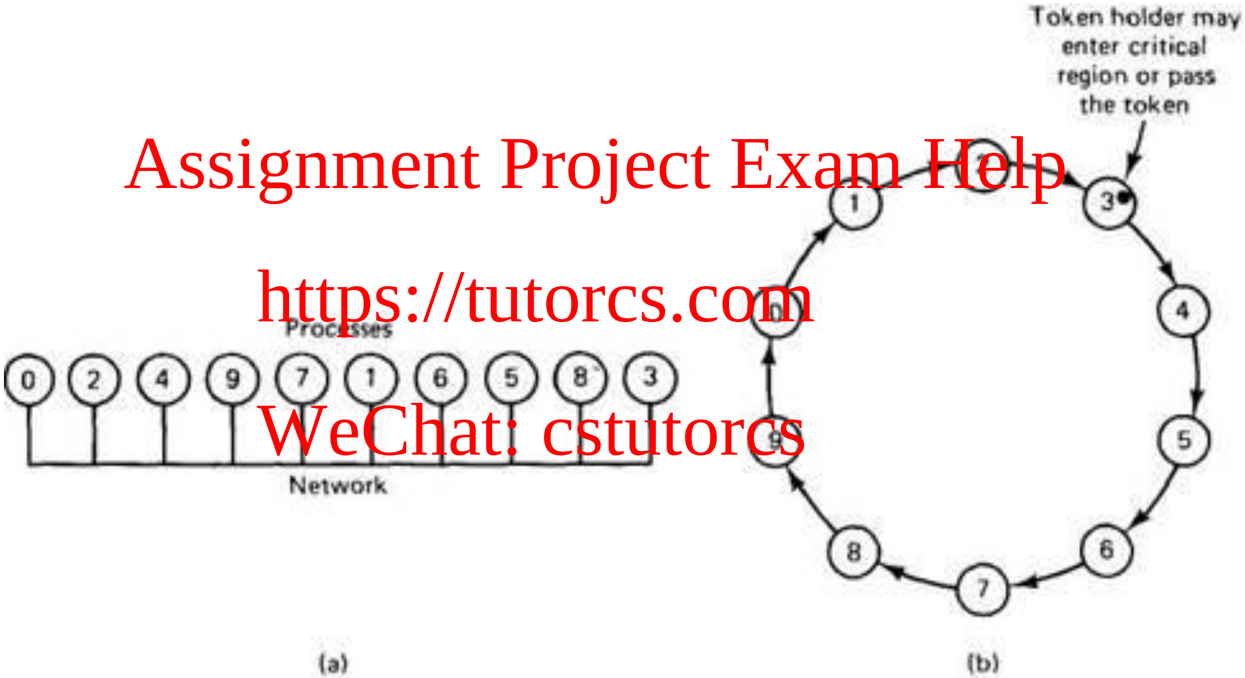


Distributed Operating Systems. Andrew Tanenbaum

# Token Ring Resource Allocation

- Also possible to use a logical token ring structure
- When the algorithm is initialised process 0 has a token
- The process passes the ring to its neighbour in the token ring structure
- When the process receives the token it checks to see if it wants the resource
- If it does it utilises the resource and then releases the token
- A process will have to wait until every other process in the token utilises the resource in the worst case scenario

# Token Ring Resource Allocation



Token holder may enter critical region or pass the token

Distributed Operating Systems. Andrew Tanenbaum

# Comparison of Resource Allocation Algorithm

| Algorithm | Messages per Resource Allocation | Delay before Resource Allocation (in message times) | Potential Problems |
|---|---|---|---|
| Centralised | 3 | 2 | Coordinator Crash (Central Point of Failure) |
| Distibuted | 2(n-1) | 2(n-1) | Crash of any process |
| Token Tring | 1 - ∞ | 0 – (n-1) | Lost token, Process Crash |

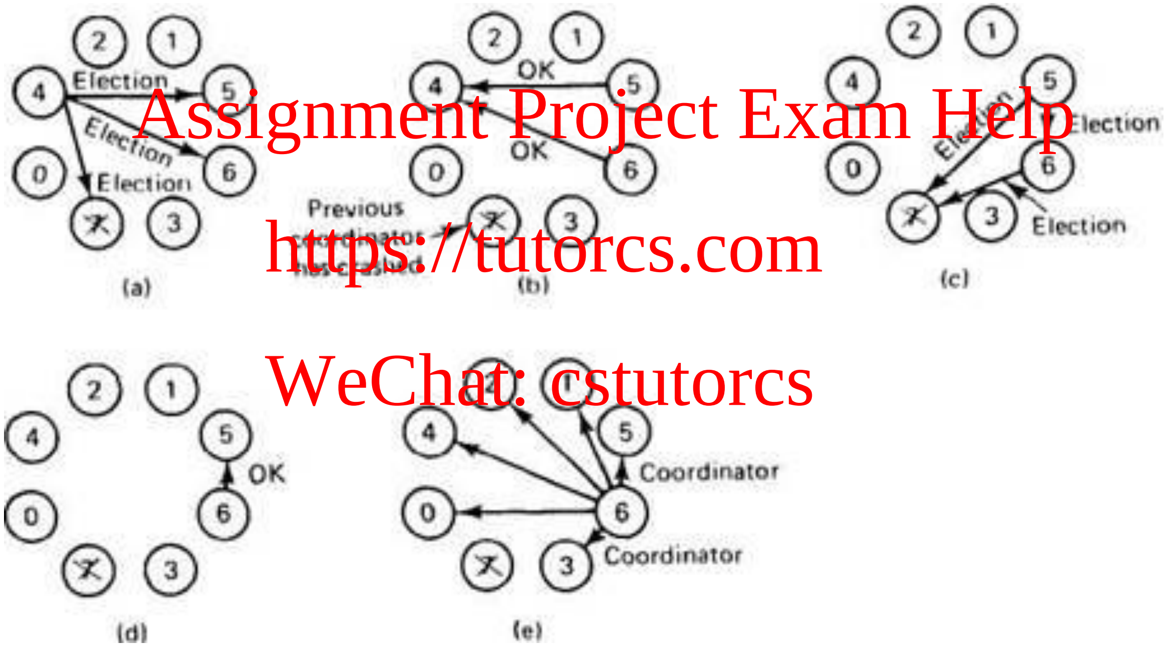
# Election Algorithms

- In the event of failure of a leader process in a centralized scheme and for other algorithms such as the Berkley Algorithm discussed earlier an election algorithm is necessary to select a new leader
- Election algorithms include:
  - Bully Algorithm
  - Ring Algorithm

# Bully Algorithm

- In the Bully algorithm each process is assigned a unique weight value
- If a process notices that the coordinator has stopped it sends election messages to processes which have a higher weight
- When a process receives an election message it replies to the election message. In addition if it has not sent messages to processes with a higher weight value it does so
- If the process receives an election message and it has sent messages it stops
- If after a certain time period a process has received no replies to its election message it determines that it has won the election and sends out a message to all processes indicating that it is the coordinator
- When a process recovers it will launch a new election

# Bully Algorithm

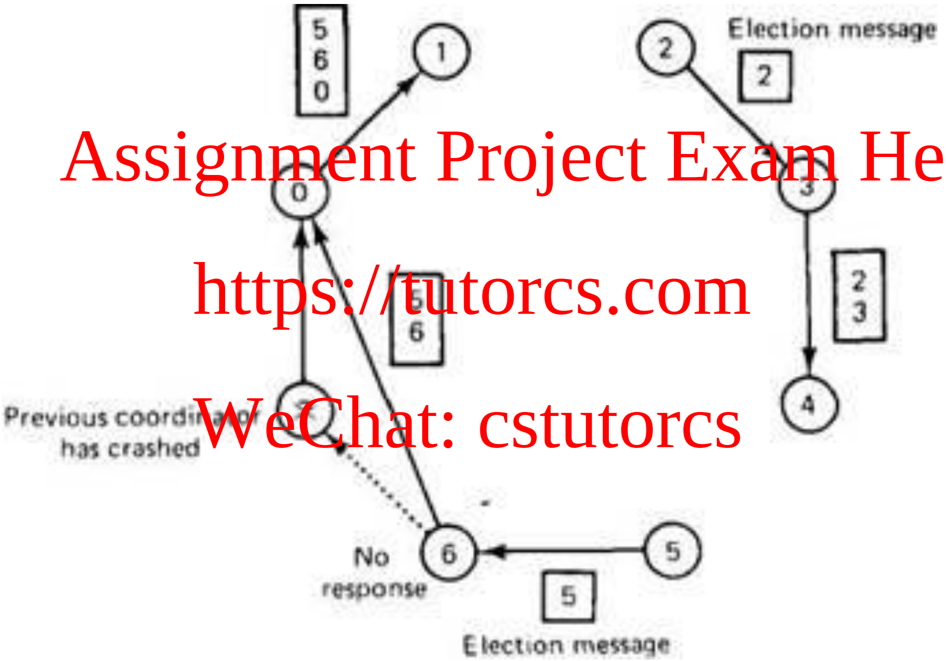


Distributed Operating Systems. Andrew Tanenbaum

# Ring Algorithm

- In a ring algorithm when any process notices that the coordinator is down it sends an election message to the next process in the logical ring
- The election message includes its process number
- If the next process in the logical ring does not acknowledge the message then the it sends it the process after this in the logical ring until it receives and acknowledgement
- When a process receives an election message it adds its process number to the election message and forwards it to the next process in the ring
- When a process receives an election message it then sends a coordinator message around the ring which indicate who is the new coordinator (the highest process number) and the members of the logical ring
- When a process receives the coordinator message it has sent it removes the message as it has gone around the ring. This done to reduce overhead

# Ring Algorithm



Distributed Operating Systems. Andrew Tanenbaum