Distributed-Computing-Module-3-Important-Topics-PYQs

- How It Works (Step-by-Step)
  - 🟢 Step 1: Asking for Permission
  - 🟠 Step 2: Receiving a Request
  - 🟡 Step 3: Using the Resource
  - 🔴 Step 4: Releasing the Resource
- 8. Compare various models of deadlock.
  - 1. Single-Resource Model
  - 2. AND Model
  - 3. OR Model
  - 4. AND-OR Model
  - 5. P-out-of-Q Model (Subset of AND-OR Model)
- 9. Illustrate Suzuki- Kasami's Algorithm.
  - Design Challenges (and How It Solves Them)
  - Correctness
  - Example
- 10. Describe how quorum-based mutual exclusion algorithms differ from the other categories of mutual exclusion algorithms.
  - How Quorum-Based Mutual Exclusion Works
  - How Quorum-Based Algorithm Differs from Other Algorithms
  - Why Quorum-Based Algorithms Are Better in Some Cases
- 11. Explain Maekawa's algorithm for mutual exclusion in detail with example
  - Key Idea
  - How It Works (Steps)
  - Example
- 12. Explain Lamport's Algorithm for Mutual Exclusion.
  - Main Idea
  - How It Works (Steps)
- 13. Explain in Detail about Deadlock handling Strategies in a Distributed environment.
  - 1. Deadlock Prevention
  - 2. Deadlock Avoidance
  - 3. Deadlock Detection

1. Explain the issues in Deadlock Detection.

1. Detection of Deadlocks

To detect a deadlock, the system needs to:

- **Maintain a Wait-For Graph (WFG)**: This graph shows which process is waiting for which other process.
- **Search for Cycles in the WFG**: A cycle means a deadlock. But in a **distributed system** (with multiple machines), the WFG is spread across many sites, so detecting cycles becomes tricky.

Correctness Conditions for Deadlock Detection

Any good deadlock detection algorithm must follow two rules:

1. **Progress**:
   - It **must detect real deadlocks** in **finite time** (not keep waiting).
   - Once a deadlock happens, detection should **keep moving** and not pause.
2. **Safety**:
   - It **must not report false deadlocks** (also called **phantom deadlocks**).
   - In distributed systems, machines don't share a global memory or clock, so sometimes they see **outdated or incomplete info**. This can make them think there's a deadlock when there isn't one.

2. Resolution of a Detected Deadlock

Once a deadlock is found, we need to fix it:

- **Break the cycle** by rolling back one or more processes.
- This frees up the resources and lets other blocked processes continue.

Also, **remove old wait-for information** from the system immediately after breaking the deadlock. If we don't, the system might wrongly detect deadlocks that **no longer exist** (again, false deadlocks).

2. List the requirements of Mutual Exclusion Algorithm.

Mutual exclusion means **only one process can be in the Critical Section (CS) at a time**. To make sure this works smoothly, a mutual exclusion algorithm must follow three main rules:

1. Safety Property

- **Only one process** can be in the **Critical Section** at any moment.
- This avoids conflicts when multiple processes try to access shared resources (like files or variables).

2. Liveness Property

- Ensures the system **doesn't get stuck**.
- **No deadlock**: Processes don't wait forever for each other.
- **No starvation**: Every process will eventually get its turn — no one is ignored forever.

> 💡 Imagine you're waiting in line. Liveness ensures the line keeps moving and no one gets stuck or skipped forever.

3. Fairness

- Every process should get a **fair chance** to enter the CS.
- Requests should be handled in the **order they arrive**, often tracked using a **logical clock** (a way to order events in distributed systems).

> 💡 Like a ticket system at a bakery — whoever takes a ticket first gets served first.

3. Calculate the rate at which a system can execute the critical section requests if the synchronization delay and average critical section execution times are 3 and 1 second respectively.

To calculate the **rate at which a system can execute critical section (CS) requests**, we consider two key components:

- **Synchronization delay**: Time needed to coordinate entry and exit into the critical section (e.g., acquiring and releasing locks, waiting for other processes, etc.)

- **Critical Section execution time**: Actual time a process spends inside the CS.

Given:

- Synchronization delay = **3 seconds**
- Critical Section execution time = **1 second**

Total time per CS request=Synchronization delay+CS execution time=3+1=4 seconds

Or, in **requests per minute**:
60/4 = 15 requests per minute

4. List any three performance metrics of mutual exclusion algorithms.

- **Message Complexity**
  - The **number of messages** needed for one process to enter the **Critical Section (CS)**.
  - Lower message count = better efficiency.
- **Synchronization Delay**
  - The **time gap** between one process **leaving** the CS and the **next one entering** it.
  - Smaller delay = faster switching between processes.
- **Response Time**
  - The **waiting time** for a process after it has sent its request to enter the CS **until it actually gets in**.
  - Shorter response time = quicker access to the CS.

5. Compare Token based approach and Non-token based approach.

Token-based Approach

- A special **token** (like a VIP pass) is passed between processes.
- **Only the process with the token** can enter the Critical Section (CS).

- It's fast and uses fewer messages, but if the token is **lost**, it must be recreated.

Non-token-based Approach

- No token is used.
- A process asks **permission from all other processes** before entering CS.
- Uses more messages, but there's **no risk of token loss**.

Feature	Token-based Approach	Non-token-based Approach
🔑 **Main Idea**	A **unique token** (privilege) is passed around.	**Permission messages** are exchanged among all sites.
🧠 **How Access is Given**	A process **enters CS only if it has the token**.	A process **enters CS if it receives permission from all others**.
📬 **Message Complexity**	**Lower** (usually ~1 to N messages per request).	**Higher** (typically 2N – 1 messages or more).
🔁 **Failure Impact**	If the token is **lost**, special handling is required to regenerate it.	If a process **crashes**, it might block others waiting for replies.
⚖️ **Fairness**	Can be made fair by passing token in a logical order.	Fairness depends on algorithm logic and may need extra care.
⚡ **Speed (Efficiency)**	Often **faster**, especially under high load.	**Slower**, due to more message exchanges.
✅ **Mutual Exclusion Guarantee**	Ensured by having **only one token**.	Ensured because only **one site's condition becomes true**.

6. Explain how wait for graph can be used in Deadlock Detection.

Imagine you and your friends are passing around books. You can only **borrow a book** if the person who has it is **done with it and returns it**. But what if **everyone is waiting for someone else to return a book**? No one can proceed—this is a **deadlock**!

In a **distributed system**, processes (computers) request resources (like files, memory, or database access). If a process is **waiting for another process** to release a resource, we can

represent this situation using a **Wait-For Graph (WFG)**.

What is a Wait-For Graph (WFG)?

A **Wait-For Graph (WFG)** is a **visual way to check for deadlocks** in a system. It is a **directed graph** where:

- **Nodes** represent **processes** (P1, P2, P3, etc.).
- **Edges (arrows)** represent **waiting relationships** (e.g., P1 → P2 means **P1 is waiting for P2** to release a resource).

How is WFG Used for Deadlock Detection?

1. **Build the Graph**:
   - Each process that is **waiting for a resource** is **connected** to the process holding that resource.
2. **Check for Cycles** 🔄:
   - If there is a **cycle (loop) in the graph**, it means a **deadlock** has occurred.
   - Example of a cycle:
     - `P1 → P2 → P3 → P1 (Deadlock! 🚨)
   - If **no cycle** is found, the system is **deadlock-free**.

Example: Detecting Deadlock Using WFG

Scenario:

- **P1 is waiting for a resource held by P2**
- **P2 is waiting for a resource held by P3**
- **P3 is waiting for a resource held by P1**

WFG Representation:

- `P1 → P2 → P3 → P1 (Cycle detected! Deadlock ❌)`
- Since there's a **cycle**, no process can proceed, confirming a **deadlock**.

---

7. Explain and Illustrate Ricart-Agrawala algorithm for achieving mutual exclusion.

Think of **mutual exclusion** like waiting in line to use an ATM. Only **one person** can use it at a time. Similarly, in a **distributed system**, multiple computers (processes) may need to access a shared resource (like a file or a database), but only **one should access it at a time**.

The **Ricart–Agrawala Algorithm** helps computers **take turns** fairly and efficiently. Here's how it works:

How It Works (Step-by-Step)

🟢 Step 1: Asking for Permission

- If a **process (computer) wants to use a resource**, it sends a **"REQUEST" message** to all other processes.
- This message has a **timestamp** (like writing down the time you arrived in line).

🟠 Step 2: Receiving a Request

- When another process receives a REQUEST, it checks:
  - **"Am I using the resource?"**
    - If **No** → It sends back **"REPLY" (Yes, you can use it)** immediately.
    - If **Yes** → It **waits and sends REPLY later** after it's done.
  - If it's also **waiting** for the resource, it compares timestamps:
    - **Smaller timestamp (arrived first)** → That process gets priority.
    - **Larger timestamp (arrived later)** → It waits.

🟡 Step 3: Using the Resource

- The process **enters the critical section (CS)** (uses the resource) **only after receiving REPLYs from all other processes**.
- This ensures only **one process uses it at a time**.

🔴 Step 4: Releasing the Resource

- Once it's done, it sends **REPLYs** to any waiting processes.
- The next process in line can now enter the CS.

---

8. Compare various models of deadlock.

Deadlock occurs when **processes wait indefinitely** for resources that are held by other processes. Distributed systems allow different ways to request resources, leading to different **models of deadlock**.

1. Single-Resource Model

- A process can request **only one resource at a time**.
- The system grants the resource only if it is available.
- If a process is already holding a resource, it must **release it before requesting another**.

Deadlock Detection:

- In a **Wait-For Graph (WFG)**, each node can have at most **one outgoing edge**.
- **If a cycle is present**, a deadlock has occurred.

Example:

- Process A requests **Resource 1**.
- Process B requests **Resource 2**.
- If **A is waiting for B and B is waiting for A**, a cycle forms → **Deadlock!**

2. AND Model

- A process can request **multiple resources at the same time**.
- The request is granted **only if all requested resources are available**.
- The requested resources can be on **different locations (servers)**.

Deadlock Detection:

- In a WFG, each node can have **multiple outgoing edges**.
- If a **cycle** exists, a deadlock has occurred.

Example:

- A process requests **Printer AND Disk Space**.
- If one is available but the other is not, the process **waits indefinitely**.

3. OR Model

- A process requests **multiple resources**, but it only needs **any one** of them to proceed.

- If at least **one resource is granted**, the process **continues execution**.

Deadlock Detection:

- A cycle in the WFG **does not always mean a deadlock** because a process may still proceed if **one** of the requested resources is available.

Example:

- A process requests **"Printer OR Scanner"**.
- If **at least one** is available, the request is satisfied.
- If neither is available, the process **waits until one is free**.

4. AND-OR Model

- A combination of the **AND model and OR model**.
- A process can request **some resources together (AND) while having options for others (OR)**.

Example:

- A process requests **"Disk AND (Printer OR Scanner)"**.
- It must get **Disk**, but it can proceed with **either a Printer or a Scanner**.
- Deadlock depends on the **specific resource combinations** and availability.

5. P-out-of-Q Model (Subset of AND-OR Model)

- A process requests **any P resources out of Q available resources**.
- If at least **P resources are available**, the request is granted.

Example:

- A cloud server needs **any 2 out of 5 available CPU cores** to process a request.
- If at least **2 cores are available**, execution proceeds.
- If fewer than **2 cores are available**, the process waits.

9. Illustrate Suzuki- Kasami's Algorithm.

Requesting the critical section:

(a) If requesting site S_i does not have the token, then it increments its sequence number, $RN_i[i]$, and sends a REQUEST(i, sn) message to all other sites. ("sn" is the updated value of $RN_i[i]$.)

(b) When a site S_j receives this message, it sets $RN_j[i]$ to $max(RN_j[i], sn)$. If S_j has the idle token, then it sends the token to S_i if $RN_j[i] = LN[i] + 1$.

Executing the critical section:

(c) Site S_i executes the CS after it has received the token.

Releasing the critical section: Having finished the execution of the CS, site S_i takes the following actions:

(d) It sets $LN[i]$ element of the token array equal to $RN_i[i]$.

(e) For every site S_j whose i.d. is not in the token queue, it appends its i.d. to the token queue if $RN_i[j] = LN[j] + 1$.

(f) If the token queue is nonempty after the above update, S_i deletes the top site i.d. from the token queue and sends the token to the site indicated by the i.d.

- **Requesting the Token**
  - If a process wants to enter the CS and **doesn't have the token**, it **broadcasts a REQUEST** to all other processes.
- **Receiving the Token**
  - The process **holding the token** checks the request.
    - If it's **not using** the CS → it **sends the token** immediately.
    - If it's **in CS** → it **sends the token after finishing**.
- **Entering the CS**
  - Once the requesting process **gets the token**, it **enters the CS**.
- **After CS Execution**
  - The process checks **which other processes have requested** the token and **sends it to the next one in line**.

Design Challenges (and How It Solves Them)

1. **Outdated Requests**
   - Each REQUEST has a **sequence number** to tell if it's new or old.
   - This avoids sending the token to a process that no longer needs it.
2. **Tracking Requests**
   - A **request queue or table** is used to track which processes have asked for the token and whether their request is still pending.

Correctness

- **Mutual Exclusion is guaranteed** because:
  - Only **one token exists**.
  - Only the **token holder can enter the CS**.

Example

Let's say we have **4 sites**: P1, P2, P3, P4

- P2 wants to enter CS → it doesn't have the token
- P2 sends **REQUEST** to P1, P3, and P4
- P4 has the token and is not in CS → it sends the token to P2
- P2 enters CS

10. Describe how quorum-based mutual exclusion algorithms differ from the other categories of mutual exclusion algorithms.

Mutual exclusion ensures that **only one process can enter the critical section (CS) at a time**. Different algorithms achieve this in **different ways**.

How Quorum-Based Mutual Exclusion Works

- Instead of **asking all sites** for permission (like in Lamport or Ricart-Agrawala algorithms), a process **only asks a subset of sites** called a **quorum**.
- **Quorums are designed to overlap**, so at least **one site** knows about both requests and ensures only **one process** enters the CS at a time.
- A site **locks its quorum members** before executing the CS.
- It **must receive a RELEASE message** before granting permission to another process.

🔹 **Example:**

Imagine you need **approval from a group of teachers** to submit an assignment. Instead of asking **all teachers**, you **ask only a small group**, ensuring **at least one teacher** is in multiple groups to avoid conflicts.

How Quorum-Based Algorithm Differs from Other Algorithms

Feature	Lamport's & Ricart-Agrawala	Quorum-Based Algorithm
Requesting Permission	Requests from **all sites**	Requests from **a subset (quorum)**
Conflict Resolution	**All sites participate**	**Only quorum members participate**
Message Complexity	**High** (all sites exchange messages)	**Lower** (fewer messages sent)
Reply Mechanism	Can send multiple replies	**One reply at a time** (locks quorum)
Efficiency	**More messages = More delay**	**Less communication = Faster execution**

Why Quorum-Based Algorithms Are Better in Some Cases

✅ **Reduces Message Complexity** – Instead of contacting all sites, only a subset is involved.
✅ **Faster Execution** – Since fewer messages are exchanged, CS is accessed quicker.
✅ **Scalable** – Works better in large distributed systems compared to Lamport's or Ricart-Agrawala.

11. Explain Maekawa's algorithm for mutual exclusion in detail with example

Requesting the critical section:

(a) A site S_i requests access to the CS by sending REQUEST(i) messages to all sites in its request set R_i.

(b) When a site S_j receives the REQUEST(i) message, it sends a REPLY(j) message to S_i provided it hasn't sent a REPLY message to a site since its receipt of the last RELEASE message. Otherwise, it queues up the REQUEST(i) for later consideration.

Executing the critical section:

(c) Site S_i executes the CS only after it has received a REPLY message from every site in R_i.

Releasing the critical section:

(d) After the execution of the CS is over, site S_i sends a RELEASE(i) message to every site in R_i.

(e) When a site S_j receives a RELEASE(i) message from site S_i, it sends a REPLY message to the next site waiting in the queue and deletes that entry from the queue. If the queue is empty, then the site updates its state to reflect that it has not sent out any REPLY message since the receipt of the last RELEASE message.

Maekawa's algorithm is a **quorum-based approach** where **each process doesn't need to get permission from all processes**, but only from a **selected group** (called a *request set*). This makes it more efficient.

Key Idea

- Each site (process) needs to get **permission from only a subset of sites** before entering the **Critical Section (CS)**.
- These subsets (called **request sets**, Ri) are designed so that **any two request sets share at least one common site**.
- This shared site ensures **no two processes can enter CS at the same time**.

How It Works (Steps)

1. **Request Phase**
   - A process sends a **REQUEST** message to all sites in its **request set (Ri)**.
2. **Grant Phase**
   - Each site can **grant REPLY** to only one request at a time.
   - If it has already granted a REPLY, it **queues the new request**.
3. **Execution Phase**

- When a process **gets all REPLYs** from its request set, it enters the CS.

4. **Release Phase**
   - After leaving the CS, the process sends **RELEASE** messages to all sites in Ri.
   - These sites then **grant REPLY** to any pending requests in their queues.

Example

Let's say we have **4 processes (P1, P2, P3, P4)**.
Suppose their **request sets** are:

- R1 = {P1, P2}
- R2 = {P2, P3}
- R3 = {P3, P4}
- R4 = {P4, P1}
  Now, if **P1** wants to enter CS:
- It sends REQUEST to P1 and P2.
- If both grant REPLY → P1 enters CS.
  Meanwhile, if **P3** tries to enter CS:
- It sends REQUEST to P3 and P4.
- If **P2** has already replied to P1 and is still waiting for RELEASE, **P3 can’t proceed** — this ensures mutual exclusion.

12. Explain Lamport's Algorithm for Mutual Exclusion.

Requesting the critical section

- When a site S_i wants to enter the CS, it broadcasts a REQUEST(ts_i, i) message to all other sites and places the request on $request\_queue_i$. ((ts_i, i) denotes the timestamp of the request.)
- When a site S_j receives the REQUEST(ts_i, i) message from site S_i, it places site S_i's request on $request\_queue_j$ and returns a timestamped REPLY message to S_i.

Executing the critical section

Site S_i enters the CS when the following two conditions hold:

L1: S_i has received a message with timestamp larger than (ts_i, i) from all other sites.

L2: S_i's request is at the top of $request\_queue_i$.

Releasing the critical section

- Site S_i, upon exiting the CS, removes its request from the top of its request queue and broadcasts a timestamped RELEASE message to all other sites.
- When a site S_j receives a RELEASE message from site S_i, it removes S_i's request from its request queue.

Main Idea

- Each request is assigned a **timestamp** using a **logical clock**.
- All requests are stored in a **queue**, sorted by timestamp.
- The process at the **top of the queue** is allowed to enter the CS.

How It Works (Steps)

1. **Requesting CS**
   - A process sends a **REQUEST(timestamp, processID)** to **all other processes**.
   - It also adds this request to its own **local queue**.
2. **Receiving a REQUEST**
   - Each process **adds the request** to its queue.
   - Then it sends back a **REPLY** to the sender.
3. **Entering CS**
   - A process enters the CS only when:
     - Its request is at the **top of its local queue**, **AND**

- It has received **REPLY messages from all other processes**.

4. **Releasing CS**
   - After CS execution, the process:
     - Removes its request from the queue.
     - Sends a **RELEASE** message to all.
5. **Receiving RELEASE**
   - Each process **removes** the released request from its local queue.

13. Explain in Detail about Deadlock handling Strategies in a Distributed environment.

In distributed systems, **deadlock** means that processes are **stuck forever**, waiting for resources held by each other.

There are **three main strategies** to handle deadlocks:

1. Deadlock Prevention

- **Idea:** Stop deadlocks from even happening.
- **How?**
  - Force processes to **request all resources at once** before starting.
  - Or, **take away resources** from other processes if needed.
- **Problem:** Too strict and inefficient in distributed systems.
  - Not practical because:
    - You may not know all resources a process will need.
    - Taking resources back from a running process is hard.
    - It wastes system resources

2. Deadlock Avoidance

- **Idea:** Allow resource requests only **if it’s safe** (i.e., won’t lead to a deadlock).
- **How?**
  - Before granting a resource, the system checks if the **global state** is safe.
  - If granting the request might cause deadlock → **don't give the resource**.

- **Problem:**
  - Needs **complete knowledge** of the system's current state (which is hard in distributed systems).
  - Messages may arrive late, and the system state can change quickly
  - So this is also **impractical in distributed systems**.

3. Deadlock Detection

- **Idea:** Let deadlocks happen, but **detect and fix them**.
- **How?**
  - The system **analyzes the process-resource graph** to check for **cycles** (which mean deadlock).
  - If found, it **kills or rolls back** one or more processes to fix it.
- **Why Best?**
  - Works well in distributed systems.
  - No need for full knowledge of future requests.
  - Practical, even with message delays and distributed state.