Deadlock is a situation in which a set of processes becomes permanently blocked, waiting for an event that only another process in the set can cause. It often happens when processes are competing for limited resources.

Key Concepts of Deadlock:

1. **Request, Allocate, Release Process**:
    - **Request**: A process requests resources (e.g., printers, memory) needed for execution.
    - **Allocate**: If the resource is available, the system allocates it to the requesting process.
    - **Release**: Once the process has finished using the resource, it releases it back to the system for other processes.

Example: Imagine two tape drives (T1 and T2) as resources. If process P1 holds T1 and process P2 holds T2, but both need the other drive to proceed, they are stuck waiting for each other, leading to a deadlock.

Deadlock Definition:

- Deadlock is a situation where a group of processes is blocked permanently, each waiting for resources that are held by others in the group. The system cannot allocate the requested resources, and processes do not proceed.

Example:

- **Resources**: Tape drives T1 and T2.
- **Processes**: P1 holds T1, P2 holds T2. Both P1 and P2 request the other tape drive, creating a circular dependency where neither process can proceed.

Types of Resources:

1. **Reusable Resources**: Can be used by only one process at a time but are not consumed. Examples include memory, processors, printers, etc.
    - Example: A printer can be used by one process at a time, but after printing, it becomes available for another process.
2. **Consumable Resources**: These resources can be produced and consumed, like signals, messages, or interrupts.
    - Example: Once an interrupt is used, it cannot be reused until another one is generated.

Communication Deadlocks:

This occurs when processes are waiting for messages from each other, but no messages are sent. All processes remain blocked.

Example:

- If processes P1, P2, and P3 are all waiting for messages from each other and no message is being sent, deadlock occurs.

Conditions for Deadlock:

For a deadlock to happen, four conditions must be met simultaneously:

1. **Mutual Exclusion**: Only one process can use a resource at a time. If a resource is being used by a process, other processes must wait.
2. **Hold and Wait**: Processes hold onto resources while waiting for additional resources.
3. **No Preemption**: Resources cannot be forcibly taken away from a process; they can only be released voluntarily.
4. **Circular Wait**: A circular chain of processes exists where each process is waiting for a resource held by the next process in the chain.

Example of Deadlock Conditions:

- **Mutual Exclusion**: Process P1 holds T1, and no other process can use it.
- **Hold and Wait**: P1 waits for T2, which is held by P2, while still holding T1.
- **No Preemption**: T1 and T2 cannot be taken away; P1 and P2 must release them voluntarily.
- **Circular Wait**: P1 waits for T2, P2 waits for T1—forming a circular chain.

Deadlock Modeling:

Deadlocks can be represented using **directed graphs**. These graphs show the relationships between processes and resources.

1. **Nodes**: Represent processes or resources.
2. **Edges**: Represent requests or assignments of resources.

- A **cycle** in the graph indicates a deadlock situation.

Example Graph:

- A cycle (P1 → R1 → P2 → R2 → P1) shows a deadlock where P1 is waiting for a resource held by P2, and P2 is waiting for a resource held by P1.

Resource Allocation Graph:

- A **Resource Allocation Graph** is a directed graph with two types of nodes: processes and resources. There are two types of edges:
    1. **Request Edges**: Directed from a process to a resource, indicating the process has requested the resource.
    2. **Assignment Edges**: Directed from a resource to a process, indicating the resource has been allocated to the process.

Example:

- If P1 requests R1 and R1 is assigned to P2, the graph has a request edge from P1 to R1 and an assignment edge from R1 to P2.

Summary:

Deadlock occurs when processes are stuck in a circular wait for resources. The four conditions—mutual exclusion, hold and wait, no preemption, and circular wait—must all be present. Deadlocks can be modeled with directed graphs, and deadlock prevention involves ensuring one of the conditions does not hold.

Deadlock

A deadlock occurs in a system when a set of processes becomes permanently blocked because each process is waiting for a resource that is held by another process in the set. Deadlocks can happen in systems with multiple processes competing for finite resources, and certain conditions must hold for a deadlock to occur.

Deadlock Conditions

Deadlock happens when all four of the following conditions are satisfied:

1. **Mutual Exclusion**: A resource can be held by only one process at a time.
2. **Hold-and-Wait**: A process holding at least one resource can request additional resources without releasing the ones it holds.
3. **No Preemption**: Resources cannot be forcibly taken from a process; they must be released voluntarily.
4. **Circular Wait**: A circular chain of processes exists, where each process is waiting for a resource held by the next process in the chain.

Resource Allocation and Deadlock Example

Consider two processes, P1 and P2, and two tape drives, T1 and T2. If P1 is allocated T1 and P2 is allocated T2, and they both request the other's resource, the system enters a deadlock state where neither process can continue.

Resource Types

- **Reusable Resources**: These resources can be used by one process at a time and are not depleted. Examples include processors, I/O channels, memory, and files.
- **Consumable Resources**: These resources are created and destroyed (consumed), such as interrupts, messages, and signals.

Deadlock Modeling

Deadlocks can be modeled using a **Resource Allocation Graph** (RAG), where:

- Nodes represent processes and resources.
- Edges represent requests and allocations of resources.
- A **cycle** in the graph is a necessary condition for deadlock, and if all resources involved have a single unit, it is also sufficient.

Wait-For Graph (WFG)

A simplified version of the RAG, the **Wait-For Graph**, eliminates the resource nodes and shows only the processes waiting for other processes. WFGs are particularly useful for modeling communication deadlocks.

Deadlock Avoidance

Deadlock avoidance requires advanced knowledge of processes' resource requirements and attempts to avoid unsafe states, which may lead to deadlock. Key strategies include:

1. **Process Initiation Denial**: Do not start a process if it may lead to deadlock.
2. **Resource Allocation Denial (Banker's Algorithm)**: Do not grant a resource request if it could lead to an unsafe state.

Safe and Unsafe States

- **Safe State**: There exists a sequence of process execution in which all processes can complete without deadlock.
- **Unsafe State**: No such sequence exists.

Deadlock Prevention

Deadlock prevention techniques aim to ensure that at least one of the four necessary conditions for deadlock does not hold. Some methods include:

1. **Collective Requests**: Prevents the hold-and-wait condition by ensuring processes request all resources at once.
2. **Ordered Requests**: Prevents circular wait by enforcing an ordering on resource requests.
3. **Preemption**: Allows resources to be forcibly taken from processes.

By following these strategies, deadlocks can be either avoided or prevented in a concurrent processing system.

Deadlock Concepts and Handling Strategies

Necessary and Sufficient Conditions for Deadlock:

1. **Cycle in the Graph:**

- If a cycle forms in the resource allocation graph, and all resources in the cycle have only one unit each, the cycle is both necessary and sufficient for a deadlock.
- **Example:** Process P1 holding resource R2 while requesting R1, and process P2 holding resource R1 while requesting R2.

2. **Knot Representation:**
    - When the resource types in the cycle have more than one unit, a cycle is necessary but not sufficient for a deadlock. In such cases, a **knot** is the sufficient condition.
    - **Example:** P1, P2, and P3 each requesting different resources in a way that creates a resource knot.

Wait-for Graph (WFG):

- A simplified version of the resource allocation graph where resource nodes are removed, and edges are collapsed. It shows which processes are waiting for other processes.
- WFGs help model communication deadlocks and detect cycles among processes.

Deadlock Handling in Distributed Systems:

Three strategies can be applied to manage deadlocks:

1. **Avoidance**: Carefully allocate resources to prevent deadlock by avoiding unsafe states.
2. **Prevention**: Imposes constraints on how resources are requested to prevent deadlock from happening.
3. **Detection and Recovery**: Allows deadlock to occur, detects it, and then resolves it.

Deadlock Avoidance:

- **Safe State**: A system is in a safe state if there exists some process sequence such that every process can finish its execution.
- **Unsafe State**: No safe sequence exists, which means a deadlock could occur.

Process Initiation Denial (Deadlock Avoidance Approach):

- A process is only started if the maximum claim of all current processes and the new one can be met, avoiding any potential deadlock.

Resource Allocation Denial (Banker's Algorithm):

- Allocating resources incrementally, ensuring that granting resources won't lead to a deadlock by analyzing the current state of the system (resources allocated, available, etc.).

Deadlock Prevention:

1. **Collective Requests**:

    - A process must request all its resources at once before it starts, preventing the **hold-and-wait** condition.

2. **Ordered Requests**:
    - Assigning a unique number to each resource type and ensuring processes request resources in strictly increasing order.

3. **Preemption**:
    - Some resources, such as CPU registers, can be easily saved and restored, allowing the system to preempt a resource and give it to another process.

Deadlock Detection and Recovery:

- Deadlocks are allowed to occur, and a **deadlock detection algorithm** is used to identify deadlocks. After detection, the system resolves the deadlock through resource preemption or terminating processes.

Global Wait-For Graph (WFG) Construction in Distributed Systems:

- Construct the resource allocation graph for each site, convert it into a WFG, and combine the local WFGs from all sites to form a global WFG.
- **Problem**: Maintaining a WFG in a distributed system can be challenging due to the need for coordination between sites.

Deadlock Detection Techniques:

1. **Centralized Approach**:
    - Each site has a local coordinator maintaining its WFG. A central coordinator then collects the local WFGs to detect global deadlocks.

2. **Hierarchical Approach**:
    - The system is divided into regions, and each region has a coordinator to handle deadlocks locally. Higher-level coordinators handle inter-region deadlocks.

3. **Distributed Approach**:
    - No central coordinator exists. All sites collaboratively participate in detecting and resolving deadlocks.

Transaction-Based Deadlock Prevention:

- Uses **unique priority numbers** for transactions (often based on timestamps) to resolve conflicts.
- **Wait-die** and **Wait-wound** schemes help decide whether a transaction should wait or abort based on its priority. In the wait-die scheme, the older transaction waits, while younger ones abort. In the wait-wound scheme, younger transactions are allowed to wait, while older ones force the younger to abort.

In summary, deadlock handling strategies in distributed systems involve a combination of avoidance, prevention, detection, and recovery techniques. These methods rely on WFGs, resource allocation policies, and sometimes, transaction-based prioritization to prevent and resolve deadlocks.

Deadlock Detection in Distributed Systems

Centralized Deadlock Detection

- **Method**: A central coordinator maintains the Wait-For Graph (WFG) for deadlock detection.
- **Information Transfer Methods**:
    - **Continuous Transfer**: Messages are sent whenever an edge is added or deleted.
    - **Periodic Transfer**: Data is sent at regular intervals.
    - **Transfer-on-Request**: Data is sent only when requested by the central coordinator.

Drawbacks of Centralized Approach:

1. Vulnerability to failure of the central coordinator.
2. Performance bottleneck in large systems.
3. Possibility of detecting false deadlocks due to incomplete or delayed information.

Example: Centralized Deadlock Detection (Steps Overview):

1. Processes P1, P2, P3, and resources R1, R2, R3.
2. Each process requests and waits for resources, forming dependencies.
3. WFGs are updated at the central coordinator.
    - If messages arrive out of order, a **phantom deadlock** may be detected.

Solution:

- Use a **global timestamp** for each message to avoid false deadlocks due to out-of-order message arrival.

Hierarchical Deadlock Detection

- Uses a **hierarchical tree** of deadlock detectors.
    - Each **controller** in the hierarchy manages deadlock detection within its range.
    - **Local controllers** manage the WFG of their site, while **higher-level controllers** manage the union of WFGs from lower-level controllers.
- **Deadlock Detection**:

- If a local WFG contains a cycle, the corresponding controller detects a deadlock and takes action.
- Cycles are not propagated to higher-level controllers unless they involve multiple sites.

Fully Distributed Deadlock Detection

- All sites share equal responsibility for detecting deadlocks.
- **Algorithms**:
    1. **WFG-based distributed algorithm**:
        - Each site maintains its own local WFG and introduces an **external process node (Pex)** for inter-site dependencies.
        - **Edges**:
            - (Pi, Pex): Pi waits for a resource at another site.
            - (Pex, Pj): Pj is waiting for a resource held at the current site.
    2. **Probe-based distributed algorithm**:
        - Uses probes to detect deadlocks across multiple sites by propagating information about process dependencies.

Example of WFG-based Fully Distributed Detection:

- If a cycle in the local WFG does not involve **Pex**, it is a **local deadlock**.
- If **Pex** is involved in the cycle, it indicates a potential **distributed deadlock**, and further detection algorithms are needed to resolve it across multiple sites.

Problem in Distributed Deadlock Detection

- **Issue**: Multiple sites may start deadlock detection simultaneously for the same processes, potentially leading to more than the necessary processes being terminated.
- **Solution**: Assign unique identifiers to each process to ensure no duplicate actions during deadlock detection.

Probe-based Distributed Deadlock Detection Algorithm

- **Proposed by**: Chandy, Mishra, Hass (CMH) in 1983.
- **Concept**: A probe message is used by the requesting process to detect deadlocks by sending a message to the process holding the requested resource.

Probe Message Structure:

1. ID of the process just blocked.
2. ID of the sender process.
3. ID of the recipient process.

How It Works:

- If the recipient is using the resource, it ignores the probe.
- If the recipient is waiting for a resource, it forwards the probe to the process holding the requested resource.
- If the probe returns to the originator, a deadlock is detected.

Features of CMH Algorithm:

- Easy to implement.
- Fixed-length messages and minimal computation.
- Low overhead without constructing graphs or gathering extensive data.
- No false deadlock detection.
- No specific structure is required among processes.

Recovery from Deadlock

1. **Operator Intervention**: Notify the operator manually to resolve the deadlock.
2. **Termination of Processes**: Terminate processes to free up resources. This requires analyzing dependencies.
3. **Rollback of Processes**: Rollback selected processes to a previous state using checkpoints to free resources.

Considerations:

- **Selection of Victim**: Based on recovery cost, starvation prevention, and impact on the system.

Election Algorithms

- **Purpose**: Select a coordinator process to manage system-wide coordination.
- **Assumption**: Each process has a unique priority number, and the process with the highest priority becomes the coordinator.

Bully Algorithm

- **Mechanism**:
    - If a process sends a request to the coordinator and receives no response, it assumes the coordinator has failed.
    - The process starts an election by sending a message to all processes with a higher priority number, informing them of the election.
    - Higher-priority processes can send an "alive" message, claiming coordination.

Features:

- **Recovery**: A previously failed coordinator can reclaim its role by sending a coordinator message.
- **Efficiency**: Requires O(n2)O(n^2)O(n2) messages in the worst case and n−2n-2n−2 in the best case.

Ring Algorithm

- **Mechanism**:
    - Processes are arranged in a logical unidirectional ring.
    - If a process notices the coordinator has failed, it starts an election by sending a message to the next process in the ring.
    - Each process adds its priority number to the election message until the highest priority is found.
    - The highest-priority process becomes the new coordinator and notifies the other processes.

Features:

- **Recovery**: A failed coordinator can rejoin by sending an inquiry to the new coordinator.
- **Drawback**: Multiple election messages may circulate simultaneously. Each election requires 2(n-1) messages, and on average, n/2 messages are required for recovery.

Summary

- **CMH Algorithm** is simple, lightweight, and efficient for deadlock detection.
- **Bully Algorithm** and **Ring Algorithm** are popular methods for electing a coordinator, each with its own benefits and limitations in terms of message complexity and ease of implementation.