1. **Deadlock Detection**

**Models of Distributed Deadlock**

In distributed systems, deadlocks can occur due to processes across multiple nodes holding resources while simultaneously waiting on others. The two main models of distributed deadlocks are:

1. **AND Model**
   1. **Description**: A process is considered deadlocked if it is waiting for all the resources it has requested to proceed. This typically occurs in tightly coupled operation scenarios where a set of resources are all needed for execution.
   * **Live Example 1:**
     + Process A holds Resource 1 and requests Resource 2.
     + Process B holds Resource 2 and requests Resource 1.
     + Circular dependency leads to deadlock.
   * **Live Example 2:**

Imagine a distributed database transaction.

* + - Process A locks Table X and requests Table Y.
    - Process B locks Table Y and requests Table X.
    - Deadlock ensues.
  1. **Live Example 3**:
     + *Scenario*: In a cloud computing environment, Process P1 on Server A holds a CPU allocation and waits for a memory slice. Process P2 on Server B holds the required memory slice and waits for a CPU slot from Server A.
     + *Result*: Both processes stall, unable to proceed without the other releasing its resources.

1. **OR Model**
   1. **Description**: A process is considered deadlocked if it is waiting for at least one of the resources it has requested to proceed. This model is more common in scenarios where multiple alternative resources can satisfy the request.
   * **Live Example 1:** 
     + In a file sharing system, multiple processes might request read or write access to the same file.
     + If one process has a read lock, and others are waiting for write locks, a deadlock occurs even though not every process needs all resources.

**[**

**Basic Scenario Description**

In a file-sharing system, files can be accessed by multiple processes for reading and writing purposes. The system uses locks to manage access to the files:

* **Read Lock (Shared Lock)**: Allows multiple processes to read the file simultaneously without any modification.
* **Write Lock (Exclusive Lock)**: Ensures that when one process is writing to the file, no other process can read from or write to the file.

**Example of Deadlock Scenario**

Consider a scenario with three processes—Process A, Process B, and Process C—and one file, File X.

1. **Process A** starts and requests a read lock on File X. The system grants this request because no write locks are held on File X.
2. **Process B** requests a write lock on File X. Since a read lock is already held by Process A, Process B must wait until all read locks are released.
3. **Process C** also requests a read lock on File X. Normally, this would be granted, but since a write lock request by Process B is pending, the system has to ensure that new read locks are not granted (to prevent further delay to the write lock request), thus Process C also must wait.
4. **Process A** continues to hold the read lock, perhaps waiting for some condition or due to a long-running read operation.

**Resulting Deadlock:**

* **Process B** is waiting for Process A (and potentially Process C) to release their read locks.
* **Process C** is waiting for the write lock request by Process B to be resolved (which depends on the release of the read lock by Process A).
* **Process A** might be unaware of B and C waiting and continues its operation under the read lock.

**Analysis**

This creates a classic deadlock situation where:

* Process B cannot proceed because it is blocked by the existing read lock held by Process A.
* Process C is also blocked, not because of Process A directly but because of the system's decision to prioritize pending write requests before allowing further reads.
* If Process A waits for some resource held by another process (not shown in the initial scenario) or is simply performing a long-running read, all processes are stuck.

**Expanded Examples**

**Example 1: Database System**

* **Processes**: Multiple database transactions.
* **Resource**: Database rows or tables.
* **Scenario**: Transaction T1 reads from Table 1; Transaction T2 attempts to write to Table 1 and is blocked. Another Transaction T3 requests to read from Table 1 but is blocked due to T2's pending write.
* **Deadlock**: T2 waits for T1 to complete; T3 cannot proceed because of T2.

**Example 2: Document Collaboration Tool**

* **Processes**: Users editing a shared document.
* **Resource**: Paragraphs or sections of the document.
* **Scenario**: User A is reading (reviewing) a section; User B wants to edit (write to) the same section and is blocked; User C wants to read the same section but is blocked due to the pending edit by B.
* **Deadlock**: User B waits for A to finish reading; User C is also waiting for B's edit to complete.

**]**

1. **Live Example 2**:
   1. *Scenario*: In a print spooling system, Process P1 is printing a document and can use any of the available printers. It currently occupies Printer A and requests either Printer B or C, both occupied.
   2. *Result*: P1 waits indefinitely if Printer B and C remain occupied, despite not needing both.
2. **Live Example 3**:
   1. *Scenario*: In a network routing algorithm, a router R1 needs to forward a packet via one of two links. Both links are controlled by routers (R2 and R3) that are waiting for other resources.
   2. *Result*: R1 is deadlocked, waiting for either link to become available, though only one is needed for the operation.
3. **Chandy-Misra-Haas Deadlock Detection**

This algorithm uses probe messages to detect deadlocks in distributed systems:

**What is a Probe?**

A probe message typically contains:

* **Initiator**: The ID of the process that started the probe.
* **Process ID**: The ID of the current process handling the probe.
* **Resources Held and Requested**: Information about what resources the current process is holding and what it is waiting for.
* **Path**: A list of process IDs that the probe has visited. This helps in tracking the probe's path across the network and detecting cycles.

**Classical Live Example: Banking System**

Consider a distributed banking system where multiple branches need to transfer funds among themselves, and each branch can initiate transactions independently. To avoid inconsistencies in balances, branches might lock certain accounts until a transaction is complete.

**Scenario Setup:**

* **Branch A**: Holds a lock on Account X and requests a lock on Account Y to complete a transaction.
* **Branch B**: Holds a lock on Account Y and requests a lock on Account Z.
* **Branch C**: Holds a lock on Account Z and requests a lock on Account X.

**Implementation of the Chandy-Misra-Haas Algorithm:**

1. **Initiation**:
   * **Branch A** starts a transaction involving Accounts X and Y but finds that Account Y is locked by **Branch B**. It sends a probe to **Branch B**.
   * The probe contains: {Initiator: A, Current: A, Resources: [X needed, Y needed], Path: [A]}
2. **Propagation**:
   * **Branch B** receives the probe, updates it, and forwards it to **Branch C**, since it is waiting for Account Z which is locked by **Branch C**.
   * Updated probe at B: {Initiator: A, Current: B, Resources: [X needed, Y held, Z needed], Path: [A, B]}
   * **Branch C** does the same, noting that it is waiting for Account X, locked by **Branch A**.
   * Updated probe at C: {Initiator: A, Current: C, Resources: [X needed, Y held, Z held], Path: [A, B, C]}
3. **Detection**:
   * **Branch C** sends the updated probe back to **Branch A** because it needs Account X.
   * **Branch A** receives a probe with its own ID as the initiator and sees that the probe has made a full circle back to it, indicating a cycle: {Initiator: A, Current: A, Resources: [X held, Y needed, Z held], Path: [A, B, C, A]}
   * Since the path contains A initially and terminates at A, a deadlock cycle is detected involving A, B, and C, all waiting on each other in a circular manner.

* **AND Model**:
  + **Initiation**: A blocked process sends a probe to all processes from which it is awaiting resources.
  + **Propagation**: When a process receives a probe, it forwards the probe to all processes it is waiting on, if it has not already done so.
  + **Detection**: If a process receives a probe that it has already seen (cycle in the probe path), a deadlock is detected.
* **OR Model:**
  + **Initiation**: Similar to the AND model, a blocked process initiates a probe.
  + **Propagation**: The probe is forwarded by any process waiting for one or more resources listed in the probe.
  + **Detection**: A deadlock is detected if a process receives a probe and the resources in the probe match any of the resources the process is currently holding.
  1. **Deadlock Resolution**

Deadlock resolution strategies can involve preventing, avoiding, detecting, and recovering from deadlocks:

1. **Prevention**:
   * **Strategy**: Modify system conditions to make deadlocks impossible.
   * **Live Example 1**: Implementing a policy where all processes must declare and request all required resources before execution begins, ensuring resources are allocated only if all are available.

**The concept highlights of preventing deadlock through a policy where all processes must declare and request all required resources before their execution begins. This approach ensures that resources are allocated only if all are available at the same time, thus preventing any process from holding part of its required resources while waiting for others. This technique is known in computer science as the "all-or-nothing" resource allocation strategy or sometimes related to the Banker's algorithm which is a variation of the deadlock avoidance algorithm**.

**Classical Live Example: Air Traffic Control System**

Scenario Setup:

Imagine an air traffic control system at a busy international airport. In this system, each airplane needs several resources to complete its landing sequence safely. These resources include runway time, gate availability, and ground crew. Given the limited availability of these resources, it's critical that they are managed meticulously to avoid conflicts (deadlocks) that could lead to delays or accidents.

Implementation of All-or-Nothing Resource Allocation:

1. **Resource Declaration**:
   * Every incoming flight must declare its resource needs in advance before beginning its landing sequence. For example, Flight A might declare a need for Runway 1, Gate 3, and two ground crew teams.
2. **Resource Allocation Check**:
   * The air traffic control system reviews all available resources and checks if Runway 1, Gate 3, and the required ground crew teams are all available simultaneously.
   * If all requested resources are available at once, they are allocated to the flight, and the landing sequence begins.
   * If any of the resources are not available (say, Gate 3 is currently occupied), then no resources are allocated to Flight A. Flight A remains in a holding pattern until all resources become available.
3. **Execution**:
   * Only when all resources have been secured does Flight A begin its landing procedure. This ensures that the flight does not start descending and then have to abort or delay landing because a resource (like a gate) is not available.

**Benefits:**

* **Safety and Efficiency**: This method ensures that flights are not committed to a landing path without having all necessary resources secured, enhancing safety and operational efficiency.
* **Deadlock Prevention**: By ensuring that no flight partially occupies key resources while waiting for others, the system prevents scenarios where multiple flights could be indefinitely delayed, a situation analogous to deadlock in computing.

Risks and Challenges:

* **Increased Waiting Times**: In some cases, flights might have to wait longer before they can begin landing procedures, especially during peak times when resources are scarce.
* **Complex Coordination**: The system requires robust, real-time communication and coordination mechanisms to track and allocate resources efficiently.

1. **Avoidance**:
   * **Strategy**: Dynamically examine the state of the system to avoid unsafe resource allocations.
   * **Live Example 1**: Using the Banker’s Algorithm, which calculates the maximum potential needs of all processes before deciding whether or not to grant a resource request.

The Banker's Algorithm, named for its similarity to the way a bank would decide whether to grant a loan, is a resource allocation and deadlock avoidance algorithm that tests for the safety by simulating the allocation for predetermined maximum possible amounts of resources before deciding whether to proceed with the transaction.

**Principles of the Banker's Algorithm:**

**Resource Claim:** Each process must a priori claim the maximum number of resources it may need. This claim must not exceed the total resources available in the system.

**Resource Allocation:** The algorithm will allocate resources to a process only if it can satisfy a safety condition after considering the maximum claims from all remaining processes.

**Safety Sequence:** The algorithm determines whether a sequence exists where all processes can finish executing without leading to a deadlock. This sequence is considered a "safe sequence."

The Banker's Algorithm is an approach to managing resource allocation and avoiding deadlocks in operating systems and other computer applications where multiple processes have competing needs for limited resources. It's named after its resemblance to the way a bank would manage its funds, making sure it can satisfy all withdrawal requests from its clients without running out of cash.

**Imagine a small system where there are two types of resources—printers and scanners.**

**System Setup**

* **Resources Available**:
  + 2 Printers
  + 1 Scanner
* **Processes**:
  + **Process A**:
    - Maximum needs: 1 printer, 1 scanner
  + **Process B**:
    - Maximum needs: 1 printer, 1 scanner

**Initial Allocation**

When the processes start, let's assume neither has been allocated any resources yet.

**Banker’s Algorithm Steps**

* **Step 1: Declare Maximum Needs** Each process declares the maximum number of each resource it may need:
* Process A: 1 printer, 1 scanner
* Process B: 1 printer, 1 scanner
* **Step 2: Resource Request** Suppose Process A requests 1 printer and 1 scanner. The system checks if granting this request leaves the system in a safe state.
* **Step 3: Check for Safe State** To check for a safe state, the system simulates whether, after granting the request, it can still satisfy the maximum possible remaining resource request from any process.
* Available after allocation to A:
  + Printers: 1 (2 total - 1 allocated to A)
  + Scanners: 0 (1 total - 1 allocated to A)
* Process B still needs 1 printer and 1 scanner at maximum, but only 1 printer is available, and no scanners are available.

**Safe State Analysis**

* **Is there a safe sequence now?** No, because if Process B requests its maximum needs now, those needs can't be met. Therefore, allocating 1 printer and 1 scanner to Process A would lead to an unsafe state.
* **Step 4: Deny or Delay Resource Allocation** Since allocating these resources to Process A results in an unsafe state, the system denies or delays the request. Process A must wait.
* **Step 5: Adjusting and Proceeding** Let's adjust and say Process A requests only 1 printer, and no scanner:
* This request is granted because it doesn't put the system into an unsafe state. After this allocation:
  + Available resources:
    - Printers: 1
    - Scanners: 1

**Safe State Re-evaluation**

* Now, both Process A and Process B's maximum requirements can still be met with the available resources, assuming that after using these resources, each process will release them once done.
* **Step 6: Proceeding with Other Requests** If Process B now requests 1 printer and 1 scanner:
* The system grants this request because even though it would use up all resources, each process has reached its maximum requirement and is expected to release the resources upon completion of its tasks.

**Conclusion**

In this example, the Banker's Algorithm helps prevent deadlock by ensuring that at no point are resources allocated in such a way that it would be impossible to meet the maximum declared needs of all processes. This cautious approach ensures that all processes can continue to execute without entering a deadlock, even if it means some processes must wait longer for resources.

1. **Detection and Recovery**:
   * **Strategy**: Allow deadlocks to occur, detect them, and recover.
   * **Live Example 1**: Implementing the Chandy-Misra-Haas algorithm to detect cycles in resource allocation graphs and then selectively abort one or more processes to break the deadlock.
   * **Live Example 2**: Utilizing timeout mechanisms where processes must complete their tasks within a certain period or be rolled back automatically if a deadlock is suspected.

Each strategy has trade-offs between simplicity, performance, resource utilization, and system throughput. The choice of strategy often depends on the specific requirements and characteristics of the system in question.