UNIT – 3

Variations of 2 phase locking mechanism/protocol

The Two-Phase Locking (2PL) protocol is a concurrency control mechanism used in database management systems to ensure serializability of transactions. It consists of two phases: the growing phase and the shrinking phase. While the basic 2PL protocol is straightforward, there are variations and enhancements that have been developed to address specific needs and situations. Here are some notable variations of the 2PL protocol:

1. **Basic Two-Phase Locking (Strict 2PL):**

   - In the basic 2PL protocol, transactions acquire locks as needed during the execution and release all locks at the end of the transaction. The strict version enforces that a transaction must acquire all the locks it needs before releasing any.

2. **Rigorous Two-Phase Locking (Rigorous 2PL):**

   - Rigorous 2PL is a more stringent form of 2PL that requires a transaction to release locks only after it has completed all its read and write operations. This can reduce the possibilities of deadlocks but may also increase contention.

3. **Wait-Die Two-Phase Locking (Wait-Die 2PL):**

   - Wait-Die 2PL is an extension of the basic 2PL that addresses deadlocks. In this variant, older transactions wait for younger transactions to release locks (wait-for graph), while younger transactions are aborted (killed) if they request a lock held by an older transaction. This prevents deadlock but may lead to transaction restarts.

4. **Wound-Wait Two-Phase Locking (Wound-Wait 2PL):**

   - Wound-Wait 2PL is another deadlock prevention mechanism where older transactions wound (abort) younger transactions if they request a lock held by a younger transaction. This variant allows older transactions to proceed but may increase the rate of aborted transactions.

5. **Deadlock Detection and Resolution with 2PL:**

   - Some systems combine the basic 2PL with deadlock detection mechanisms. Transactions can request locks as needed and, if a deadlock is detected, the system resolves it by aborting one or more transactions to break the deadlock. This approach allows for more flexibility while controlling deadlocks.

6. **Two-Phase Locking with Timeout:**

- In this variation, transactions are allowed to request locks with a specified timeout. If the lock cannot be acquired within the given time, the transaction is either aborted or put on hold. This can prevent long-lasting deadlocks but may introduce complexities related to timeout management.

7. **Strict Two-Phase Locking (Strict 2PL):**

   - Strict 2PL enforces a strict two-phase lock release policy. A transaction can only release locks after it has successfully completed all its operations. This variant can be more restrictive but simplifies deadlock detection and resolution.

8. **Shared and Exclusive Locking (Readers-Writers 2PL):**

   - In scenarios where multiple transactions need concurrent access to data, a variant of 2PL allows shared locks (for reading) and exclusive locks (for writing). Shared locks can be held simultaneously by multiple transactions, while exclusive locks are mutually exclusive.

9. **Distributed Two-Phase Locking:**

   - In distributed databases, variations of 2PL are tailored to handle distributed transactions and global lock management. These variants often incorporate communication and coordination mechanisms to ensure data consistency across multiple sites.

10. **Timestamp-Based Two-Phase Locking:**

    - Some systems combine 2PL with timestamp-based concurrency control to provide precise control over the execution order of transactions based on their timestamps, ensuring serializability.

The choice of a 2PL variant depends on specific database system requirements, the nature of transactions, and the trade-offs between contention, deadlock management, and system complexity. Each variant has its advantages and disadvantages, and the selection of the appropriate variant should be based on a careful analysis of the system's needs and constraints.

Cautious waiting, no waiting, and time out protocols in deadlock prevention

In the context of deadlock prevention, several protocols aim to avoid or mitigate the occurrence of deadlocks. Cautious waiting, no waiting, and timeout protocols are three distinct approaches to handling potential deadlocks in a multi-process or multi-transaction system:

1. **Cautious Waiting Protocol:**

   - **Definition:** Cautious waiting is a protocol that allows processes or transactions to wait for the required resources but does so in a way that minimizes the risk of deadlocks.

- **How It Works:** In this protocol, when a process or transaction requests a resource that is currently held by another process, it waits for the resource to become available, but with conditions. These conditions might involve assigning priorities to processes, ensuring that a lower-priority process does not indefinitely block a higher-priority process. Additionally, there may be a maximum waiting time for a process.

- **Impact:** Cautious waiting reduces the chances of deadlocks by allowing processes to wait for resources while controlling waiting times and priorities. However, it can still result in contention and suboptimal resource utilization.

2. **No Waiting Protocol (Resource Allocation Graph Protocol):**

- **Definition:** The no waiting protocol, often implemented using a resource allocation graph, does not allow processes or transactions to wait for resources. Instead, it employs a "request-and-release" mechanism.

- **How It Works:** When a process requires a resource that is already held by another process, the requesting process does not wait. It releases all its currently held resources, including those that are not requested by other processes. This action forces all processes involved in the deadlock to release their resources, effectively breaking the deadlock.

- **Impact:** The no waiting protocol can be effective in preventing deadlocks but may lead to frequent process interruptions and resource thrashing. It is commonly used in systems where processes cannot tolerate waiting for resources.

3. **Timeout Protocol:**

- **Definition:** The timeout protocol involves allowing processes to wait for resources but with a predefined maximum waiting time (a timeout). If a resource is not granted within this time frame, the requesting process may be forcibly terminated (aborted or rolled back).

- **How It Works:** When a process requests a resource, it is given a specific amount of time to obtain it. If the resource is not granted within this time, the process is aborted. This approach ensures that waiting processes do not indefinitely hold resources.

- **Impact:** The timeout protocol can be effective in preventing indefinite resource blocking, but it may result in process interruptions and transaction rollbacks. It is often used in distributed systems or systems with strict timing constraints.

The choice of which protocol to implement depends on the specific requirements and constraints of the system. Each protocol has its advantages and disadvantages, and the decision should consider factors such as the nature of the processes or transactions, the importance of deadlock prevention, the impact of process interruptions or transaction rollbacks, and the system's performance goals.

Deadlock and starvation

Deadlock and starvation are two related but distinct problems that can occur in concurrent and multi-process systems. They both involve issues with resource allocation and contention, but they manifest differently and have different implications:

Deadlock:

1. **Definition:** Deadlock is a situation where two or more processes or threads in a system are unable to proceed because each is waiting for a resource held by another.

2. **Cause:** Deadlock typically occurs when multiple processes or threads compete for exclusive access to resources (e.g., locks, memory, or I/O devices) and each process holds at least one resource while waiting for another that is held by another process.

3. **Resolution:** Deadlocks can be resolved using various strategies, such as process termination, resource preemption, or timeout mechanisms.

4. **Impact:** Deadlocks can bring a system to a standstill, causing processes to hang indefinitely, and they often require intervention to resolve.

Starvation:

1. **Definition:** Starvation is a condition where a process or thread is denied access to resources it needs to make progress for an extended period, while other processes continue to access and release resources.

2. **Cause:** Starvation can occur due to resource allocation policies that prioritize certain processes over others, leading to some processes repeatedly being denied the resources they need.

3. **Resolution:** Starvation can be mitigated by implementing fair resource allocation policies, such as priority-based scheduling, or by using techniques like aging to give lower-priority processes opportunities to access resources.

4. **Impact:** Starvation does not necessarily bring the entire system to a halt, but it can result in reduced performance and fairness, and it may lead to inefficiency and low resource utilization.

In summary, deadlock is a condition where processes are mutually blocked, unable to make progress, while starvation is a condition where a process is consistently denied access to resources it needs, causing it to make slow or no progress. Deadlocks are typically more critical issues because they can lead to system-wide freezes, whereas starvation can be less disruptive but still problematic in terms of resource fairness and system efficiency. Mitigating both deadlock and starvation requires careful resource allocation and scheduling strategies.

Wait-die, Wound-wait mechanisms for deadlock prevention

Wait-die and wound-wait are two deadlock prevention mechanisms used to manage concurrency and resolve potential deadlocks in a multi-process or multi-transaction system. They are designed to handle situations where processes or transactions request resources held by others. These mechanisms are primarily used in scenarios involving shared resources. Here's an overview of both mechanisms:

Wait-Die Mechanism:

1. **Definition:** Wait-Die is a deadlock prevention mechanism that differentiates between older and younger processes (or transactions) when resource requests are made. It allows older processes to wait for resources held by younger processes but prevents younger processes from waiting for resources held by older processes.

2. **How It Works:**

   - When a younger process requests a resource currently held by an older process, it is not allowed to wait. Instead, it is "killed" or aborted.

   - When an older process requests a resource held by a younger process, it is allowed to wait for the resource.

3. **Impact:**

   - Wait-Die ensures that older, more advanced processes are not disrupted by younger ones. It minimizes the risk of deadlocks but may lead to the killing of younger processes, causing potential data loss and disruption.

Wound-Wait Mechanism:

1. **Definition:** Wound-Wait is another deadlock prevention mechanism that distinguishes between older and younger processes when resource requests are made. In this mechanism, older processes are wounded (aborted) if they request a resource held by a younger process.

2. **How It Works:**

   - When a younger process requests a resource held by an older process, the older process is "wounded" (aborted) to release the resource.

   - When an older process requests a resource held by a younger process, it is allowed to wait for the resource.

3. **Impact:**

   - Wound-Wait ensures that younger processes are not disrupted by older ones. It reduces the risk of deadlocks but may lead to the aborting of older processes, which can impact data consistency and processing efficiency.

In both mechanisms, the aim is to prevent deadlock by ensuring that only one of the competing processes is allowed to wait, while the other is either killed (in Wait-Die) or wounded (in Wound-Wait) to release the resource. The choice between Wait-Die and Wound-Wait depends on system requirements and priorities.

Considerations:

- Wait-Die is more conservative as it favors older processes, while Wound-Wait is more aggressive and prioritizes younger processes.

- The selection of the mechanism should align with the specific use case and application requirements, considering factors like data consistency, process priority, and performance.

Both mechanisms address the critical issue of deadlock prevention by making decisions based on the age of processes and the resource requests they make. The primary difference between them lies in which process, the older or younger one, is given priority in accessing the resource.

Different levels of isolation

In the context of database management systems, isolation levels are a way to control the visibility of concurrent transactions' changes to the data. Different isolation levels provide varying levels of data consistency, isolation, and concurrency. The standard isolation levels defined by the SQL standard are:

1. **Read Uncommitted (Isolation Level 0):**

   - In this isolation level, transactions can see uncommitted changes made by other transactions. It offers the least data consistency and isolation but the highest level of concurrency.

2. **Read Committed (Isolation Level 1):**

   - In Read Committed, transactions can only see committed changes made by other transactions. Uncommitted changes are not visible. This level provides a moderate level of data consistency and concurrency.

3. **Repeatable Read (Isolation Level 2):**

   - In Repeatable Read, transactions see a consistent snapshot of the data from the beginning of the transaction. This means that any changes made by other transactions after the current transaction started are not visible. It offers a higher level of data consistency but may lead to some degree of phantom reads (i.e., rows appearing or disappearing).

4. **Serializable (Isolation Level 3):**

- Serializable provides the highest level of isolation and consistency. It ensures that transactions appear to execute one after the other, and it prevents anomalies such as dirty reads, non-repeatable reads, and phantom reads. Serializable transactions are executed in such a way that the final result is equivalent to running them one after the other, although this level can be less concurrent.

5. **Snapshot Isolation:**

- Snapshot Isolation is not a standard SQL isolation level but is supported by some database systems. In Snapshot Isolation, each transaction sees a consistent snapshot of the database at the beginning of the transaction. It is similar to Repeatable Read but allows for greater concurrency by avoiding some of the issues with phantom reads.

6. **Serializable Snapshot Isolation:**

- Serializable Snapshot Isolation is a stricter version of Snapshot Isolation. It provides full serializability, similar to Serializable, while still allowing for higher concurrency.

It's important to note that the actual behavior of these isolation levels can vary between different database management systems. Also, the choice of isolation level should be based on the specific requirements of your application. Higher isolation levels provide stronger data consistency but may result in decreased concurrency, potentially impacting performance. On the other hand, lower isolation levels allow for more concurrency but can lead to data anomalies if not managed carefully. The selection of the appropriate isolation level depends on balancing data integrity and application performance.

Concurrency

Concurrency in Advanced Database Technologies:

Concurrency control is a critical aspect of advanced database technologies, ensuring that multiple users or transactions can access and modify data simultaneously without conflicts or data inconsistencies. Here are some brief notes on concurrency in advanced database technologies, worth 15 marks:

1. **Definition of Concurrency Control**:

- Concurrency control is a database management system's (DBMS) mechanism that manages simultaneous access and manipulation of data by multiple transactions to maintain data integrity and consistency.

2. **Challenges in Concurrency** :

- As databases handle multiple users and transactions concurrently, issues such as data conflicts, race conditions, and lost updates can arise.

- The challenge is to allow concurrent access while preserving the ACID properties (Atomicity, Consistency, Isolation, Durability) of transactions.

3. **Isolation Levels** :

- Isolation levels in concurrency control define the degree to which transactions are isolated from one another. Common isolation levels include Read Uncommitted, Read Committed, Repeatable Read, and Serializable.

- Higher isolation levels offer more data consistency but may lead to reduced performance due to increased locking and resource contention.

4. **Lock-Based Concurrency Control** :

- Locking is a traditional method for concurrency control. Transactions request locks on data items to prevent other transactions from accessing them.

- Types of locks include shared locks (read access) and exclusive locks (write access). Lock contention can lead to deadlocks.

5. **Timestamp-Based Concurrency Control** :

- Timestamp-based mechanisms assign a unique timestamp to each transaction and data item. Transactions with earlier timestamps get priority.

- Older transactions might be rolled back to allow younger, more current transactions to proceed.

6. **Optimistic Concurrency Control** :

- Optimistic concurrency control assumes that conflicts are rare. Transactions are allowed to proceed without locking but are checked for conflicts at the end.

- If a conflict is detected, the system rolls back and retries the transaction.

7. **Multiversion Concurrency Control** :

- Multiversion concurrency control maintains multiple versions of a data item to allow different transactions to read and write without blocking.

- It's often used in systems that require high read concurrency, like snapshot isolation in some relational databases.

8. **Advantages** :

- Enables high levels of concurrent access, improving system throughput.

- Ensures data consistency and integrity by preventing conflicting transactions from affecting each other.

9. **Disadvantages** :

- Overhead: Concurrency control mechanisms introduce overhead in terms of processing and memory.

- Complexity: Implementing and configuring concurrency control can be complex, and poorly designed schemes may lead to performance bottlenecks.

In advanced database technologies, efficient concurrency control is essential for supporting modern applications and workloads that demand high levels of simultaneous data access and modification. Different concurrency control mechanisms are chosen based on the specific requirements and trade-offs of the application and the underlying database system.

Lock-based concurrency control

Lock-based concurrency control is a method used in database management systems and concurrent programming to control access to shared resources, such as database records, files, or memory locations, in a way that ensures data consistency and prevents data corruption in a multi-user or multi-threaded environment. Locks are used to synchronize access to these resources, allowing multiple transactions or threads to work concurrently while preventing conflicting interactions. Here are the key components and concepts related to lock-based concurrency control:

1. **Locks:**

- Locks are mechanisms that control access to resources by allowing only one transaction or thread to acquire a lock on a resource at a time. Locks can be of different types, including shared locks and exclusive locks.

2. **Shared Locks:**

- Shared locks (read locks) allow multiple transactions or threads to read the resource simultaneously. They are used when multiple users can access the resource concurrently without conflicts.

3. **Exclusive Locks:**

   - Exclusive locks (write locks) ensure that only one transaction or thread can modify a resource at any given time. They are used to protect resources that must not be accessed simultaneously by multiple users to maintain data integrity.

4. **Lock Modes:**

   - Locks can have different modes, including shared, exclusive, and intention locks. Intention locks signal the intent to acquire other locks in a hierarchy.

5. **Lock Manager:**

   - The lock manager is a component of the database or concurrency control system responsible for maintaining a record of which resources are locked and by whom.

6. **Lock Compatibility Matrix:**

   - A lock compatibility matrix defines which combinations of lock types are compatible. For example, shared locks are usually compatible with other shared locks but not with exclusive locks.

7. **Deadlock Detection and Resolution:**

   - Lock-based concurrency control systems often include mechanisms to detect and resolve deadlocks. Deadlocks occur when multiple transactions are each waiting for a resource held by another. Techniques like timeout or deadlock detection algorithms are used to resolve these situations.

8. **Lock Granularity:**

   - Lock granularity refers to the level at which locks are acquired. Granularity can vary from fine-grained (e.g., individual data items) to coarse-grained (e.g., entire tables or files). Fine-grained locking can lead to higher concurrency but can introduce complexity.

9. **Lock Escalation:**

   - Lock escalation is a process in which a database management system may convert multiple fine-grained locks into fewer, coarser-grained locks to reduce overhead when too many fine-grained locks are held.

10. **Lock Timeouts:**

- Lock-based concurrency control systems may use lock timeouts to prevent transactions or threads from waiting indefinitely for a lock. If a lock cannot be acquired within a specified time, the requesting process is either aborted or put on hold.

Lock-based concurrency control is widely used in database systems and concurrent programming because it provides a structured way to manage access to shared resources, ensuring data consistency and preventing data corruption. However, it can introduce complexities related to deadlocks, performance overhead, and lock contention, which require careful management and tuning in practice.