JAVA 8 LOCKS

Optimistic And Pessimistic Locking

1. Optimistic Locking

Optimistic locking assumes that conflicts between transactions or threads are rare. It allows multiple readers (or transactions) to access the shared resource simultaneously without locking it initially, believing that conflicts are unlikely to occur. If a conflict does happen (such as when a write is attempted on the same data being read), the system detects it and rolls back or retries the transaction.

Key Features of Optimistic Locking:

- **Conflict Check on Commit**: An optimistic lock typically verifies at the end of an operation whether a conflict occurred, usually by checking a version or timestamp.
- **Low Contention Scenarios**: Optimistic locking is ideal in environments with a high read-to-write ratio, where most transactions are non-conflicting reads.
- **Implementation**: Commonly implemented in database systems by checking if a record's version or timestamp has changed during a transaction.

Example Scenario for Optimistic Locking:

Imagine an e-commerce application where multiple users frequently read product details but only occasionally modify them (e.g., updating stock after a purchase). With optimistic locking:

1. A user reads a product's details without locking it.
2. When updating stock, the system checks if the data (like a timestamp) changed since it was first read.
3. If the data hasn't changed, the update proceeds; if it has, the system aborts the update and asks the user to retry, avoiding conflicts.

Benefits:

- **Higher Concurrency**: Since multiple readers can access the resource simultaneously, optimistic locking reduces contention, allowing higher read concurrency.
- **Reduced Locking Overhead**: The absence of locks on reads means less overhead in accessing the data.

Drawbacks:

- **Conflict Handling**: If there are frequent writes or updates, the chance of conflicts increases, which can lead to repeated retries and reduce performance.

2. Pessimistic Locking

Pessimistic locking assumes that conflicts are likely and proactively prevents them by locking the resource as soon as it is accessed. A lock is placed immediately, blocking other threads or transactions from accessing the resource until the lock is released. This approach is more restrictive but ensures that conflicts are avoided during the operation.

Key Features of Pessimistic Locking:

- **Immediate Lock Acquisition**: When a resource is accessed (for reading or writing), it is locked immediately, preventing other operations from proceeding until the lock is released.
- **Ideal for High-Conflict Scenarios**: Pessimistic locking works well in environments with frequent writes or high contention on resources, as it strictly controls access.
- **Implementation**: Commonly implemented in database systems using SELECT ... FOR UPDATE statements to lock rows for updates.

Example Scenario for Pessimistic Locking:

In a banking application, where a user's account balance is frequently updated by various transactions (deposits, withdrawals), pessimistic locking can ensure consistency:

1. When a transaction starts to update the balance, it locks the account record.
2. Other transactions attempting to access the same account are blocked until the lock is released.
3. Once the update is complete, the lock is released, allowing other transactions to proceed.

Benefits:

- **Data Integrity**: Since only one thread or transaction can access the locked resource, data consistency is strictly maintained.
- **Better in Write-Heavy Workloads**: Pessimistic locking prevents data inconsistency in systems with frequent write operations.

Drawbacks:

- **Reduced Concurrency**: Since resources are locked as soon as they are accessed, other transactions must wait, reducing the potential concurrency.
- **Higher Overhead**: Managing locks adds overhead, particularly in systems with frequent reads and low conflict, as many threads may need to wait for locks to be released.

Summary Comparison

Feature	Optimistic Locking	Pessimistic Locking
Conflict Handling	Detects conflicts on commit/validation	Prevents conflicts by acquiring locks early
Concurrency	High, ideal for read-heavy workloads	Lower, suitable for write-heavy workloads
Use Case	Low write contention scenarios	High contention or frequent write scenarios
Overhead	Low on reads, may retry on conflict	High due to lock management and blocking

Use Cases

- **Optimistic Locking** is ideal in scenarios with:
  - High read and low write operations.
  - Low contention for the same resources.
  - Applications where retries are acceptable and data consistency can be checked upon commit (e.g., e-commerce catalogs, user profile views).
- **Pessimistic Locking** is ideal in scenarios with:
  - High write operations or high contention.
  - Applications where conflicts must be prevented at all costs (e.g., banking systems, inventory management).

Both locking mechanisms are valuable in managing concurrency in multithreaded applications and databases, and the choice between them depends on the application's concurrency needs and tolerance for data contention.

Intrinsic And Extrinsic Locks in Java 8

In Java, intrinsic and extrinsic locks are two types of locking mechanisms to control access to shared resources in a multi-threaded environment:

1. Intrinsic Locks (Synchronized Locks)

- Also known as **monitor locks**, these are locks provided by Java's `synchronized` keyword.
- Intrinsic locks are automatically associated with every Java object.
- When a method or block is marked with `synchronized`, the associated intrinsic lock (the object's monitor lock) is acquired by the thread before entering the synchronized code.
- Only one thread can hold the intrinsic lock of an object at a time, making it easy to prevent concurrent access.

Example of intrinsic locking:

```
public synchronized void criticalSection() {
        // Only one thread can execute this method at a time
    }

    public void anotherCriticalSection() {
        synchronized(this) {
            // Only one thread can execute this block at a time
        }
    }
```

Deadlock situation in synchronized lock

A **deadlock** in synchronized locks occurs when two or more threads are blocked forever, each waiting for the other to release a lock. This situation typically arises when multiple threads hold locks on resources and request locks on other resources, leading to a circular dependency.

Deadlock Example with synchronized Blocks

Consider two resources, ResourceA and ResourceB, and two threads, Thread-1 and Thread-2:

1. Thread-1 locks ResourceA and tries to lock ResourceB.
2. Meanwhile, Thread-2 locks ResourceB and tries to lock ResourceA.

This situation creates a circular wait, as both threads are waiting for the other to release a lock.

2. Extrinsic Locks (Explicit Locks)

- Extrinsic locks are not associated with the object itself but are explicitly created and managed using classes like *ReentrantLock*, *ReentrantReadWriteLock*, or *StampedLock* from the *java.util.concurrent.locks* package.
- Unlike intrinsic locks, extrinsic locks provide more control and flexibility, such as try-locking, timed waiting, and fairness policies.
- Extrinsic locks are often used for more complex locking requirements where `synchronized` does not offer enough control.

Example of extrinsic locking with ReentrantLock:

```
ReentrantLock lock = new ReentrantLock();

    public void criticalSection() {
        lock.lock();
        try {
            // Critical section
        } finally {
            lock.unlock(); // Always release the lock
        }
    }
```

Differences Between Intrinsic and Extrinsic Locks

Feature	Intrinsic Lock	Extrinsic Lock
Locking Mechanism	synchronized keyword	ReentrantLock, StampedLock
Flexibility	Limited control	Greater control (try-lock, fairness, timed lock)
Fairness Option	No	Yes (in ReentrantLock)
Condition Support	Limited (wait()/notify)	Full (Condition objects)
Performance	Simple to use, efficient in basic scenarios	Slightly more complex but more flexible

Choosing Between Intrinsic and Extrinsic Locks

- Intrinsic locks are simpler to use and are sufficient for basic thread synchronization.
- Extrinsic locks are suitable for advanced requirements, such as timed locking, interruptible locking, and fairness policies.

Types of Extrinsic Locks in Java 8

In Java 8, locks are part of the `java.util.concurrent.locks` package, which provides more advanced locking mechanisms than the built-in `synchronized` keyword. These locks offer finer control over the lock behavior and include features like fairness, reentrancy, try-locking with timeouts, and interruptible lock acquisition.

Here are some key locks in Java 8:

1. ReentrantLock

- A commonly used lock that is reentrant, meaning the thread that holds the lock can acquire it again without deadlock.
- Allows for fairness (FIFO) ordering if configured.
- Provides methods like `lock()`, `unlock()`, `tryLock()`, and `tryLock(long timeout, TimeUnit unit)` for more control.

```
ReentrantLock lock = new ReentrantLock();
lock.lock();
try {
   // critical section
} finally {
   lock.unlock();
}
```

2. ReentrantReadWriteLock

- A lock that separates read and write operations, allowing multiple readers but only one writer at a time.
- Improves performance in scenarios with more reads than writes.
- Contains `readLock()` and `writeLock()` methods, each returning a `Lock` instance.

```
ReentrantReadWriteLock rwLock = new ReentrantReadWriteLock();
rwLock.readLock().lock();
try {
   // read operations
} finally {
   rwLock.readLock().unlock();
}
rwLock.writeLock().lock();
try {
   // write operations
} finally {
rwLock.writeLock().unlock();
}
```

3. StampedLock

- Introduced in Java 8, a more sophisticated lock that supports three modes: *write lock*, *read lock*, and *optimistic read*.
- Optimistic reads don't block and are faster, but they need validation to ensure no write has occurred in the meantime.

```
StampedLock stampedLock = new StampedLock();
long stamp = stampedLock.tryOptimisticRead();
try {
    // optimistic read
    if (!stampedLock.validate(stamp)) {
        // retry or convert to a read lock if write occurred
    }
} finally {
    stampedLock.unlock(stamp);
}
```

4. Condition

- Works with `ReentrantLock` to enable thread signaling similar to `wait()`/`notify()`.
- Useful for cases where threads need to wait for specific conditions.

```
ReentrantLock lock = new ReentrantLock();
Condition condition = lock.newCondition();
lock.lock();
try {
    condition.await(); // await the condition
    // critical section
    condition.signalAll(); // signal all waiting threads
} finally {
    lock.unlock();
}
```

Each of these locking mechanisms is useful in specific scenarios, providing control over thread behavior and improving concurrency performance.

How ReentrantLock lock solves deadlock?

The `ReentrantLock` class in Java provides more advanced and flexible locking mechanisms than the `synchronized` keyword, allowing developers to avoid or handle deadlocks more effectively in several ways. While `ReentrantLock` doesn't inherently prevent deadlocks, it offers features that help reduce the likelihood or manage the risk of deadlock situations.

Here's how `ReentrantLock` can help manage deadlocks:

1. Try-Locking with Timeout

The `tryLock(long timeout, TimeUnit unit)` method allows a thread to attempt to acquire a lock within a specific timeout. If the lock cannot be acquired within this period, the thread can abandon the attempt, preventing it from waiting indefinitely and potentially avoiding deadlock.

This is helpful in scenarios where multiple locks need to be acquired in sequence, as it allows the program to retry or execute alternative logic if the required locks are unavailable.

Example:

1. Thread-1 attempts to acquire `lockA` and then `lockB` with a timeout.
2. Thread-2 tries to acquire `lockB` and then `lockA` with a timeout.
3. If either thread cannot acquire the required locks within the timeout, it releases any locks it holds and abandons its attempt, preventing a deadlock.

2. Interruptible Locking

`ReentrantLock` provides the `lockInterruptibly()` method, which allows a thread to acquire the lock but also respond to interruptions. If a thread is blocked waiting for a lock, it can be interrupted, allowing it to release any locks it holds and avoid waiting indefinitely, which is especially helpful in deadlock-prone scenarios.

3. Explicit Lock Ordering

By using multiple `ReentrantLock` objects with a consistent lock acquisition order across methods (e.g., always acquiring `lockA` before `lockB`), you can avoid circular wait conditions, which are a primary cause of deadlocks. Although this approach can be applied to `synchronized` blocks, `ReentrantLock` provides more control over lock acquisition and release, making it easier to enforce order.

4. Fair Locking

`ReentrantLock` supports fairness, which ensures that locks are granted to threads in the order they are requested, reducing the chance of starvation and mitigating deadlock risks in high-contention situations. You can create a fair lock by specifying `true` when creating a `ReentrantLock`:

```
Lock fairLock = new ReentrantLock(true);
```