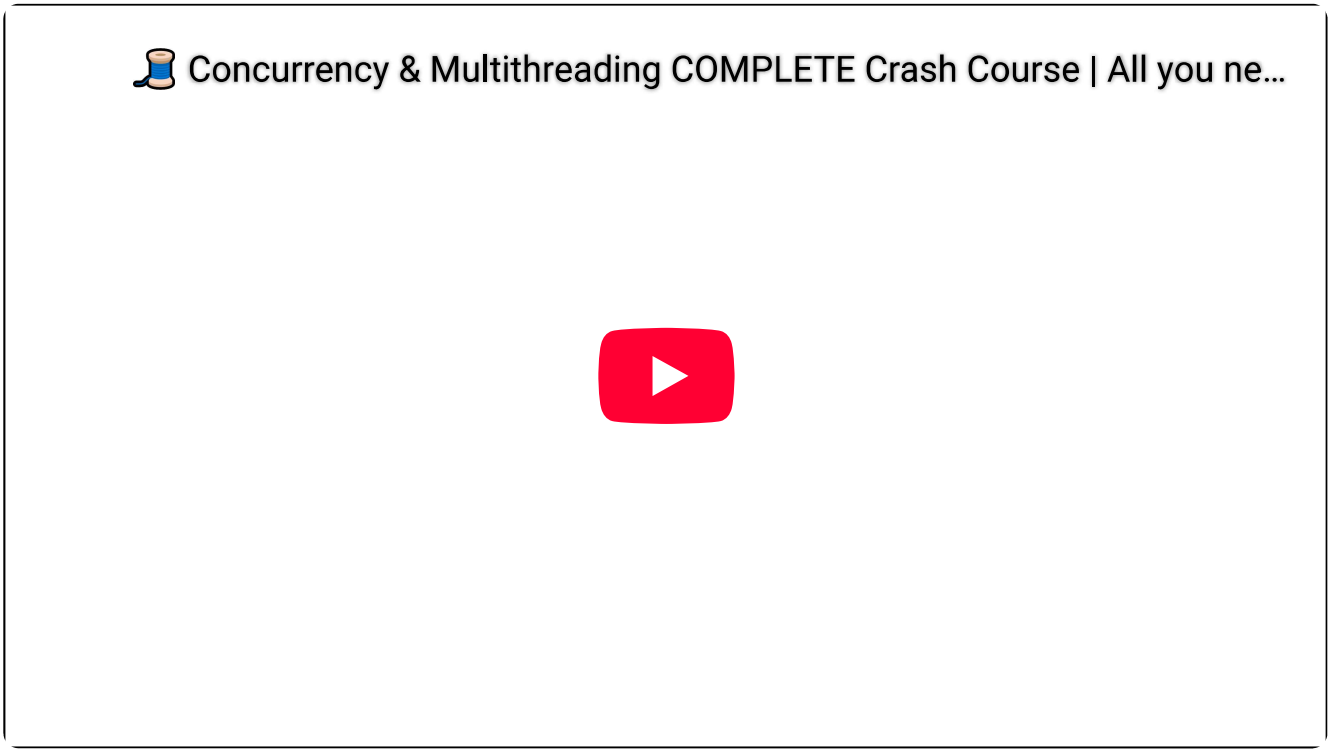
Locks and Types of Locks



Topic Tags:

system design lld

🐆⬛ **Github Codes Link**: https://github.com/aryan-0077/CWA-LowLevelDesignCode

Locks and Types of Locks in Java: Mastering Concurrency Control 🔒🔑

Lock mechanisms are essential tools for controlling access to shared resources in concurrent Java applications. While the synchronized keyword provides basic locking functionality, the java.util.concurrent.locks package offers more sophisticated and flexible lock implementations. This article explores the various types of locks available in Java and their appropriate use cases for effective concurrency control. 🚀

Why Use Explicit Locks? 🤔

The synchronized keyword has been part of Java since its inception, so why use explicit locks? Explicit locks offer several advantages:
• Greater flexibility:
Fine-grained control over lock acquisition and release 🔄

• Non-block-structured locking:
Acquire and release locks in different scopes 🔒

• Timed lock attempts:
Try to acquire a lock for a specified duration ⏱️

• Interruptible lock acquisition:
Allow threads to be interrupted while waiting for locks 🛑

• Non-ownership releases:
Release locks from different threads (with caution) 👥
• Multiple condition variables:
Associate multiple conditions with a single lock 📋

• Fairness policies:
Optional first-come-first-served lock acquisition 🎯

The Lock Interface Hierarchy 🏛️

The java.util.concurrent.locks package provides a rich set of interfaces and implementations:
1. 🔐Locks & ReentrantLock :
Locks in Java (via the Lock interface) offer more flexible and fine-grained control over synchronization than the built-in synchronized keyword. One of the most popular implementations is ReentrantLock, which is called “reentrant” because the thread that holds the lock can re-acquire it without causing deadlock 🔁.

It provides additional capabilities such as:
• ⚠️ Interruptible Lock Acquisition: Using lockInterruptibly()
• ⏳ Try-Lock Methods: With or without timeouts
• ⚖️ Fairness Policies: To ensure threads acquire locks in the order requested

It is used when you need advanced control over locking 🧠 (e.g., trying to acquire a lock and/or setting up fairness) or when a portion of a critical section is complex and may require more nuanced lock handling 🧩.

Example :

Java

```java
public class ReentrantLockExecutorExample {
  private int counter = 0;

  // Create a ReentrantLock instance.
  private final ReentrantLock lock = new ReentrantLock();

  // Method to increment the counter using the lock.
  public void increment() {
    lock.lock();
    try {
      System.out.println(Thread.currentThread().getName() + " acquired the
      counter++;
      System.out.println(Thread.currentThread().getName() + " incremented
    } finally {
      System.out.println(Thread.currentThread().getName() + " released the
      lock.unlock();
    }
  }

  public int getCounter() {
    return counter;
  }

  public static void main(String[] args) {
    ReentrantLockExecutorExample example = new ReentrantLockExecutorExampl
    // Create an ExecutorService with a fixed thread pool of 5 threads.
    ExecutorService executorService = Executors.newFixedThreadPool(5);
    // Submit 5 tasks to the executor service.
    for (int i = 0; i < 5; i++) {
      executorService.submit(() -> { example.increment(); });
    }
    // Shutdown the executor service gracefully.
    executorService.shutdown();
    try {
      // Wait for all tasks to finish; if not completed within 5 seconds,
          if (executorService.awaitTermination(5, TimeUnit.SECONDS)) {
            System.out.println("Final counter value: " + example.getCounte
          } else {
```

```
            System.out.println("Timeout: Not all tasks finished.");
        }
    } catch (InterruptedException e) {
        System.err.println("Interrupted while waiting for tasks to finis
        Thread.currentThread().interrupt();
    }
  }
}
```

OUTPUT: (Ordering May Vary)

```
pool-1-thread-1acquired the lock.
pool-1-thread-1incremented counter to: 1
pool-1-thread-1released the lock.

pool-1-thread-3acquired the lock.
pool-1-thread-3incremented counter to: 2
pool-1-thread-3released the lock.

pool-1-thread-2acquired the lock.
pool-1-thread-2incremented counter to: 3
pool-1-thread-2released the lock.

pool-1-thread-4acquired the lock.
pool-1-thread-4incremented counter to: 4
pool-1-thread-4released the lock.

pool-1-thread-5acquired the lock.
pool-1-thread-5incremented counter to: 5
pool-1-thread-5released the lock.

Final counter value: 5
```

1. ReentrantReadWriteLock (Read-Write Lock) 🔒 :
ReentrantReadWriteLock(found in the java.util.concurrent.lockspackage) divides the lock into two parts—a read lock and a write lock. It is useful when:

• Multiple Threads Need to Read: 👥 They can do so concurrently if there's no writing.

• Exclusive Writing ✍️:When a thread is updating data, no other thread (reader or writer) is allowed to access the resource.

It is used to improve performance in scenarios with many more read operations than writes.

Java

```java
public class ReadWriteLogExample {
    private final ReentrantReadWriteLock rwLock = new ReentrantReadWriteLo
    private int logValue = 0;

    // Simulate processing work using a dummy computation loop.
    private void simulateWork() {
        long sum = 0;
        for (int i = 0; i < 500000; i++) {
            sum += i;
        }
        // (The computed sum is discarded; its purpose is solely to consum
    }

    // Write operation: exclusively updates the shared logValue.
    public void writeValue(String taskName, int newValue) {
        rwLock.writeLock().lock();
        try {
            System.out.println(taskName + " (write): Acquired write lock."
            simulateWork();
            logValue = newValue;
            System.out.println(taskName + " (write): Updated logValue to "
        } finally {
            System.out.println(taskName + " (write): Released write lock."
            rwLock.writeLock().unlock();
        }
    }

    // Read operation: reads the shared logValue.
    public void readValue(String taskName) {
        rwLock.readLock().lock();
        try {
            System.out.println(taskName + " (read): Acquired read lock. Re
            simulateWork();
            System.out.println(taskName + " (read): Finished reading.");
        } finally {
            System.out.println(taskName + " (read): Released read lock.");
            rwLock.readLock().unlock();
```

```
        }
    }

    public static void main(String[] args) {
        ReadWriteLogExample logExample = new ReadWriteLogExample();
        // Create an ExecutorService with a fixed pool of 4 threads.
        ExecutorService executor = Executors.newFixedThreadPool(4);
        /*
         - Schedule tasks to simulate the following sequence:
         - 1. Start with three reader tasks concurrently.
         - 2. Then, a writer task updates the log.
         - 3. Next, two readers read the updated value.
         - 4. Then, a second writer task updates the log.
         - 5. Finally, one more reader reads the new value.
         */
        // Submit three concurrent reader tasks.
        executor.submit(() -> logExample.readValue("Reader-2"));
        executor.submit(() -> logExample.readValue("Reader-3"));

        // Submit a writer task.
        executor.submit(() -> logExample.writeValue("Writer-1", 100));

        // Submit two additional reader tasks.
        executor.submit(() -> logExample.readValue("Reader-4"));
        executor.submit(() -> logExample.readValue("Reader-5"));

        // Submit a second writer task.
        executor.submit(() -> logExample.writeValue("Writer-2", 200));

        // Submit a final reader task.
        executor.submit(() -> logExample.readValue("Reader-6"));

        // Shut down the executor.
        executor.shutdown();
        try {
            if (!executor.awaitTermination(10, TimeUnit.SECONDS)) {
                System.out.println("Timeout waiting for tasks to finish.")
            }
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
```

```
    }
}
```

Output: (Ordering May Vary)

```
Reader-1 (read): Acquired read lock. Reading logValue: 0
Reader-2 (read): Acquired read lock. Reading logValue: 0
Reader-3 (read): Acquired read lock. Reading logValue: 0
Reader-1 (read): Finished reading.
Reader-1 (read): Released read lock.
Reader-2 (read): Finished reading.
Reader-2 (read): Released read lock.
Reader-3 (read): Finished reading.
Reader-3 (read): Released read lock.

Writer-1 (write): Acquired write lock.
Writer-1 (write): Updated logValue to 100
Writer-1 (write): Released write lock.

Reader-4 (read): Acquired read lock. Reading logValue: 100
Reader-5 (read): Acquired read lock. Reading logValue: 100
Reader-4 (read): Finished reading.
Reader-4 (read): Released read lock.
Reader-5 (read): Finished reading.
Reader-5 (read): Released read lock.

Writer-2 (write): Acquired write lock.
Writer-2 (write): Updated logValue to 200
Writer-2 (write): Released write lock.

Reader-6 (read): Acquired read lock. Reading logValue: 200
Reader-6 (read): Finished reading.
Reader-6 (read): Released read lock.
```

What is the difference between synchronized and Reentrant Lock? 🔄🔒

1. Acquisition and Flexibility: 🚦

synchronized:

• The synchronized keyword is built into the language; it automatically acquires and releases the intrinsic lock (monitor) of an object. 🔄

• It is simple to use but offers only blocking behavior—it always waits indefinitely to acquire the lock. ⏳

• You cannot try to acquire a synchronized lock with a timeout or check if the lock is available (i.e., no non-blocking acquisition). ❌

ReentrantLock:
• Part of the java.util.concurrent.lockspackage, ReentrantLock provides explicit lock management. 🛠️
• It gives you extra flexibility—for instance, with methods such as tryLock()(with or without a timeout) you can attempt to acquire the lock in a non-blocking manner. ⏱️
• It also supports interruptible lock acquisition (lockInterruptibly()) and fairness policies. ⚖️

2. Automatic vs. Manual Release: 🔄 VS 🤲
• synchronized:
• The lock is automatically released when the synchronized block or method exits (even if an exception occurs). 🔄
• ReentrantLock:
• You must explicitly call unlock()(usually in a finallyblock) to ensure that the lock is released. This gives you additional control but also adds responsibility. 🔑

Java

```java
public class ReentrantLockTryLockExample {
    private final ReentrantLock lock = new ReentrantLock();
    // Task that holds the lock for an extended period.
    public void longTask(String taskName) {
        lock.lock();
        try {
            System.out.println(taskName + " acquired the lock and is perfo
            // Simulate a long operation (e.g., by sleeping or doing busy
            // Here, we sleep to emulate that long operation.
            Thread.sleep(5000);
            System.out.println(taskName + " finished the task and is relea
        } catch (InterruptedException e) {
            System.out.println(taskName + " was interrupted.");
            Thread.currentThread().interrupt();
        } finally {
            lock.unlock();
        }
    }

    // Task that attempts to acquire the lock using tryLock with a timeout
    public void tryLockTask(String taskName) {
        try {
            // Try to acquire the lock for 2 seconds.
            if (lock.tryLock(2, TimeUnit.SECONDS)) {
```

```
                try {
                    System.out.println(taskName + " acquired the lock usin
                } finally {
                    lock.unlock();
                }
            } else {
                System.out.println(taskName + " could not acquire the lock
            }
        } catch (InterruptedException e) {
            System.out.println(taskName + " was interrupted while waiting
            Thread.currentThread().interrupt();
        }
    }
    public static void main(String[] args) {
        ReentrantLockTryLockExample example = new ReentrantLockTryLockExam
        // Use ExecutorService to manage threads.
        ExecutorService executor = Executors.newFixedThreadPool(2);
        // Submit Task-A to acquire the lock and hold it for a long time.
        executor.submit(() -> example.longTask("Task-A"));
        // Short delay to ensure Task-A acquires the lock first.
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
        // Submit Task-B that attempts to acquire the lock using tryLock.
        executor.submit(() -> example.tryLockTask("Task-B"));
        // Shutdown the executor.
        executor.shutdown();
    }
}
```

Output :

```
Task-A acquired the lock and is performing a long task.
Task-B could not acquire the lock using tryLock within 2 seconds.
Task-A finished the task and is releasing the lock.
```

Key Takeaways 💡

synchronized:

• Would force Task-B to wait indefinitely until Task-A releases the lock. ⏳
• Lacks the non-blocking or timed acquisition option. ❌

ReentrantLock:

• With tryLock(long time, TimeUnit unit), Task-B can attempt to acquire the lock but proceed (or take alternate action) if it's not available within a specified timeout. ⏱️
• Offers greater flexibility and control over lock acquisition and release. 🎛️

Conclusion 🎯

Locks are fundamental tools for managing concurrent access to shared resources in Java applications. While the synchronized keyword provides basic locking functionality, the java.util.concurrent.locks package offers more sophisticated and flexible options to address complex concurrency challenges.

By understanding the different types of locks and their appropriate use cases, developers can create more efficient, scalable, and robust concurrent applications.

As concurrent programming becomes increasingly important in modern software development, mastering these lock mechanisms becomes essential for writing high-performance, thread-safe applications. 🚀🔒