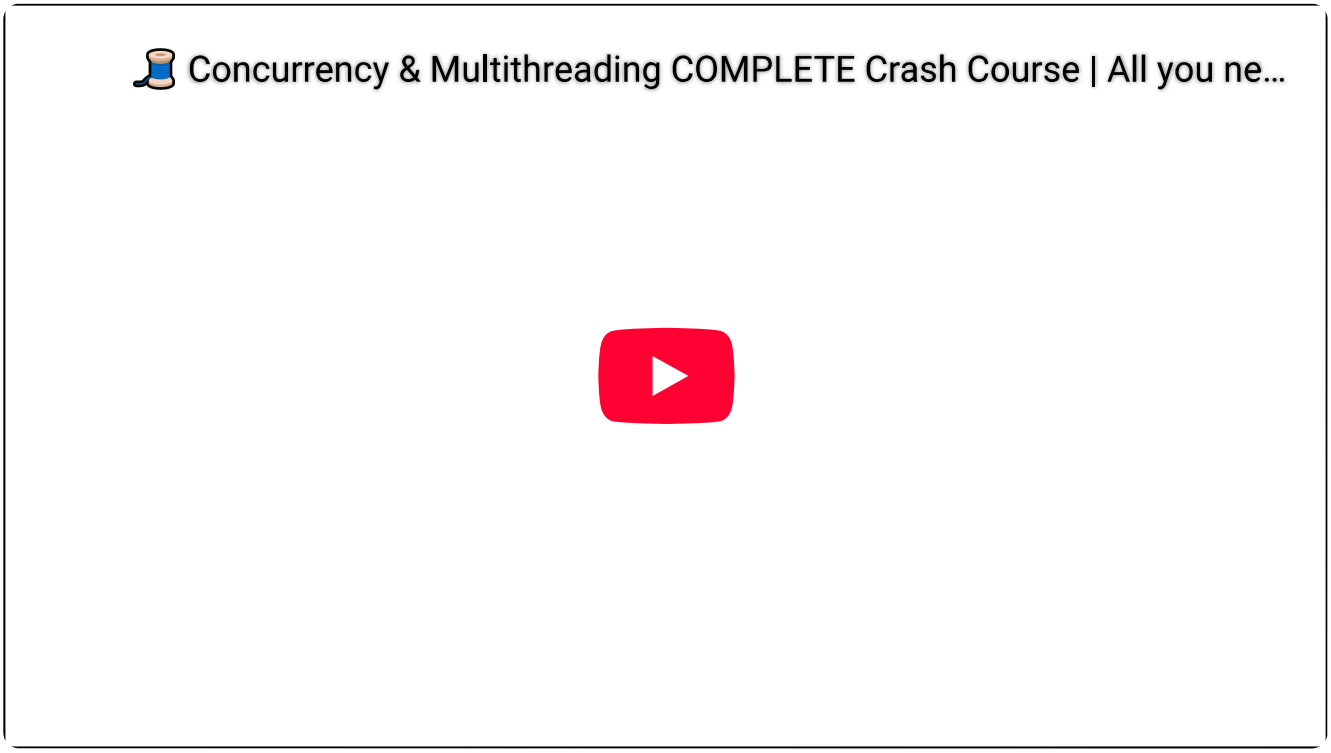
Search articles...

Sign In

Thread Synchronization



Topic Tags:

lld system design

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

Thread Synchronization in Java

Thread synchronization is a critical concept in multithreaded programming that ensures multiple threads access shared resources in a controlled manner. Proper synchronization prevents data corruption, race conditions, and ensures thread safety in a concurrent environment.

Methods of Thread Synchronization:

The synchronized keyword ⭐ is used to control access to critical sections of code so that only one thread can execute the synchronized code at a time. This ensures that shared mutable data is not corrupted by concurrent modifications ⚠️🎚️.

There are two common ways to achieve this:

1️⃣ Synchronized Method 🔄

When you declare an entire method as synchronized, the lock 🤏 is acquired on the object instance (or on the Class object for static methods) before the method is executed and released after it finishes ✅.

📌 This is useful when the whole method represents a critical section where no concurrent execution is desired. It is straightforward and reduces the chance of forgetting to protect part of the code 🧠🧩.

Example:

Java

```java
public class CounterSyncMethod {
    private int count = 0;
    // The entire method is synchronized.
    public synchronized void increment() {
        System.out.println("Synchronized Method - Start increment: " + Thr
        // Critical section: updating the shared counter
        count++;
        System.out.println("Synchronized Method - Counter value after incr
        System.out.println("Synchronized Method - End increment: " + Threa
    }

    public int getCount() {
        return count;
    }

    // Main method to test the synchronized method
    public static void main(String[] args) {
        CounterSyncMethod counter = new CounterSyncMethod();
        int numberOfThreads = 5;
        Thread[] threads = new Thread[numberOfThreads];
        // Create and start threads that call the synchronized increment m
        for (int i = 0; i < numberOfThreads; i++) {
            threads[i] = new Thread(new Runnable() {
```

```
                public void run() {
                    counter.increment();
                }
            }, "Thread-" + (i + 1));


            threads[i].start();
        }
        // Wait for all threads to complete.
        for (int i = 0; i < numberOfThreads; i++) {
            try {
                threads[i].join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        // Display the final counter value.
        System.out.println("Final counter value: " + counter.getCount());
    }
}
```

Output : The output order may vary due to thread scheduling, but it will be similar to:

```
Synchronized Method - Start increment: Thread-1
Synchronized Method - Counter value after increment: 1
Synchronized Method - End increment: Thread-1
Synchronized Method - Start increment: Thread-2
Synchronized Method - Counter value after increment: 2
Synchronized Method - End increment: Thread-2
Synchronized Method - Start increment: Thread-3
Synchronized Method - Counter value after increment: 3
Synchronized Method - End increment: Thread-3
Synchronized Method - Start increment: Thread-4
Synchronized Method - Counter value after increment: 4
Synchronized Method - End increment: Thread-4
Synchronized Method - Start increment: Thread-5
Synchronized Method - Counter value after increment: 5
Synchronized Method - End increment: Thread-5
Final counter value: 5
```

2 Synchronized Block

A synchronized block allows you to specify a particular block of code to be synchronized, along with the object on which to acquire the lock (often called a monitor). This is more fine-grained compared to a synchronized method.

✅ You can perform non-critical work outside the block, while only protecting the portion of code that truly requires exclusive access.

⚡ This can improve performance if only a subset of the method's operations need synchronization.

💡 The primary reason to choose a synchronized block over a synchronized method is when you have additional work in the method that doesn't need to be synchronized. This allows concurrent threads to execute the non-critical sections without waiting for the lock.

Example:

Java

```java
public class CounterSyncBlock {
    private int count = 0;
    // Explicit lock object for finer control.
    private final Object lock = new Object();

    public void increment() {
        // Non-critical part: runs without locking.
        System.out.println("Non-Synchronized part (pre-processing): " + Th
        // Critical section: only this part is synchronized.
        synchronized (lock) {
            System.out.println("Synchronized Block - Start increment: " +
            count++;
            System.out.println("Synchronized Block - Counter value after i
            System.out.println("Synchronized Block - End increment: " + Th
        }
        // Non-critical part: runs after the synchronized block.
        System.out.println("Non-Synchronized part (post-processing): " + T
    }

    public int getCount() {
        return count;
    }

```

```
    // Main method to test the synchronized block functionality.
    public static void main(String[] args) {
        CounterSyncBlock counter = new CounterSyncBlock();
        int numberOfThreads = 5;
        Thread[] threads = new Thread[numberOfThreads];
        // Create and start threads that execute the increment method.
        for (int i = 0; i < numberOfThreads; i++) {
            threads[i] = new Thread(new Runnable() {
                public void run() {
                    counter.increment();
                }
            }, "Thread-" + (i + 1));
            threads[i].start();
        }
        // Wait for all threads to finish.
        for (int i = 0; i < numberOfThreads; i++) {
            try {
                threads[i].join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        // Display the final value of the counter.
        System.out.println("Final counter value: " + counter.getCount());
    }
}
```

Output : The output order may vary due to thread scheduling, but it will be similar to:

```
Non-Synchronized part (pre-processing): Thread-1
Non-Synchronized part (pre-processing): Thread-2
Synchronized Block - Start increment: Thread-2
Synchronized Block - Counter value after increment: 1
Synchronized Block - End increment: Thread-2
Non-Synchronized part (post-processing): Thread-2
Non-Synchronized part (pre-processing): Thread-3
Synchronized Block - Start increment: Thread-1
Synchronized Block - Counter value after increment: 2
Synchronized Block - End increment: Thread-1
Non-Synchronized part (post-processing): Thread-1
Non-Synchronized part (pre-processing): Thread-4
```

```
Synchronized Block - Start increment: Thread-3
Synchronized Block - Counter value after increment: 3
Synchronized Block - End increment: Thread-3
Non-Synchronized part (post-processing): Thread-3
Non-Synchronized part (pre-processing): Thread-5
Synchronized Block - Start increment: Thread-4
Synchronized Block - Counter value after increment: 4
Synchronized Block - End increment: Thread-4
Non-Synchronized part (post-processing): Thread-4
Synchronized Block - Start increment: Thread-5
Synchronized Block - Counter value after increment: 5
Synchronized Block - End increment: Thread-5
Non-Synchronized part (post-processing): Thread-5
Final counter value: 5
```

🔁 Volatile Keyword in Java 🔁

The volatile keyword in Java is used to indicate that a variable's value will be modified by multiple threads 🧵🧵. Declaring a variable as volatile ensures two key things:

1️⃣ Visibility 👀

When a variable is declared volatile, its value is always read from and written to the main memory 🧠 instead of a thread's local cache.
✅ This means changes made by one thread are immediately visible to others.
❌ Without volatile, updates in one thread might not be seen (or might be delayed) by others due to caching 🔄.

2️⃣ Ordering 📏

volatile establishes a happens-before relationship 🔗.
🔐 Operations on a volatile variable cannot be re-ordered relative to each other.
📍This is especially helpful when using flags or controlling execution flow to ensure instructions are executed in the intended order.

🕹️ When to Use volatile 🧠

• Flags and Status Variables 🚩
Used to signal threads (e.g., a shutdown flag or status switch).

• Singleton Patterns (with double-checked locking) 🔒
In lazy initialization patterns, volatile ensures that the constructed instance is visible to all threads correctly.

• Lightweight Synchronization □□
If you only need visibility guarantees (not atomicity for compound actions like x++), volatile is lighter and faster than using synchronized.

Example:

Java

```java
public class VolatileExample {
    // Declaring the flag as volatile ensures that changes to 'running'
    // in one thread are immediately visible to other threads.
    private volatile boolean running = true;
    // Method executed by the worker thread.
    public void runTask() {
        System.out.println("WorkerThread: Starting execution...");
        int counter = 0;
        // Continuously increment counter until 'running' becomes false.
        while (running) {
            counter++;
        }
        System.out.println("WorkerThread: Detected stop signal. Final coun
    }
    // Called by the main thread to stop the worker thread.
    public void stopTask() {
        running = false;
    }
    // Main method to run the example.
    public static void main(String[] args) {
        VolatileExample example = new VolatileExample();
        // Create and start the worker thread.
        Thread workerThread = new Thread(new Runnable() {
            public void run() {
                example.runTask();
            }
        }, "WorkerThread");
        workerThread.start();
        // Let the worker thread run for a while.
        try {
            Thread.sleep(2000);  // Main thread sleeps for 2 seconds
        } catch (InterruptedException e) {
            e.printStackTrace();
```

```
        }
        System.out.println("MainThread: Stopping the worker thread.");
        example.stopTask(); // Signal the worker thread to stop
        // Wait for the worker thread to finish execution.
        try {
            workerThread.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("MainThread: Execution finished.");
    }
}
```

Output : the output should be similar to the following (note that the actual counter value will be a large number and can vary by run):

```
WorkerThread: Starting execution...
MainThread: Stopping the worker thread.
WorkerThread: Detected stop signal. Final counter value: 123456789
MainThread: Execution finished.
```

Atomic Variables:

Atomic variables in Java—found in the java.util.concurrent.atomic package—are designed to support lock-free 🔒, thread-safe 🛡️ operations on single variables.
You should use atomic variables when you need to perform simple operations ➕➖🔁 (like incrementing, decrementing, or updating) on shared variables in a multithreaded environment 🧵🧵. They are especially useful when the overhead of locking is undesirable 🐢 and when the logic remains limited to single-step atomic operations 🎯.

Example :

Java

```
import java.util.concurrent.atomic.AtomicInteger;
public class AtomicCounterExample {
    // The AtomicInteger counter provides atomic methods for thread-safe o
    private AtomicInteger counter = new AtomicInteger(0);

    // This method atomically increments the counter and prints the update
```

```
    public void increment() {
        int newValue = counter.incrementAndGet(); // Atomically increments
        System.out.println(Thread.currentThread().getName() + " incremente
    }

    // Retrieves the current counter value.
    public int getCounter() {
        return counter.get();
    }

    // Main method to run the AtomicCounterExample.
    public static void main(String[] args) {
        final AtomicCounterExample example = new AtomicCounterExample();
        int numberOfThreads = 10;
        // Each thread will perform 100 increments.
        int incrementsPerThread = 100;
        Thread[] threads = new Thread[numberOfThreads];
        // Create and start threads that perform increments on the atomic
        for (int i = 0; i < numberOfThreads; i++) {
            threads[i] = new Thread(new Runnable() {
                public void run() {
                    for (int j = 0; j < incrementsPerThread; j++) {
                        example.increment();
                    }
                }
            }, "Thread-" + (i + 1));
            threads[i].start();
        }
        // Wait for all threads to complete execution.
        for (int i = 0; i < numberOfThreads; i++) {
            try {
                threads[i].join();
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }
        // Display the final counter value.
        System.out.println("Final counter value: " + example.getCounter())
    }
}
```

Output : The exact interleaving of thread prints may vary on every run, but the final counter value will consistently reflect the total number of increments performed.

```
Final counter value: 1000
```

Conclusion 🎯

Thread synchronization is essential for building reliable concurrent applications. By using the appropriate synchronization mechanisms—from basic synchronized blocks to advanced utilities in the java.util.concurrent package—developers can ensure data integrity and prevent concurrency issues. 👨‍💻🔧

Understanding the trade-offs between different synchronization techniques enables you to write high-performance multithreaded code that is both correct and scalable. Remember that while excessive synchronization can lead to contention and reduced performance, insufficient synchronization can lead to subtle and hard-to-reproduce bugs. Finding the right balance is key to successful concurrent programming. 🚀⚖️