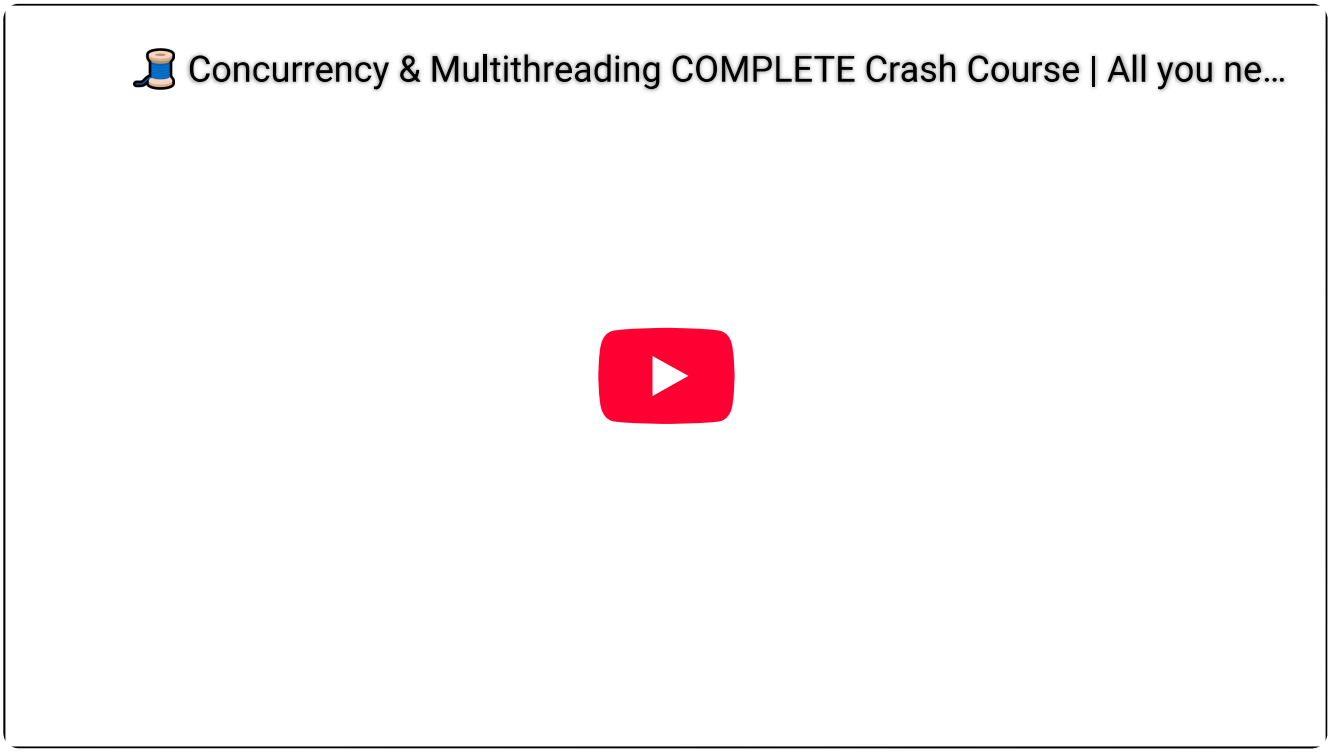
Thread Communication



Topic Tags:

system design lld

Thread Communication in Java

Thread communication is a fundamental concept in concurrent programming that allows multiple threads to coordinate and share data effectively. Proper thread communication is essential for building robust, efficient, and thread-safe applications.

Methods of Thread Communication:

wait(), notify(), and notifyAll() Methods:

> *These methods work with a thread's monitor (the intrinsic lock on an object) to coordinate the execution between threads. They are used when one or more threads need to wait for a specific condition to occur while another thread notifies them of the change.*

1. ⏸️ **wait():**

When a thread calls the wait() method on an object, it releases the monitor (lock) it holds on that object and goes into a waiting state.

> 🔁 *Use wait() when a thread needs to pause execution until some condition (usually represented by a shared variable) changes. For example, a consumer thread might wait for a producer to produce an item.*

2. 🔔 **notify():**

The notify() method wakes up a single thread that is waiting on the object's monitor. If more than one thread is waiting, the scheduler chooses one arbitrarily.

> 🎯 *Use notify() when only one waiting thread needs to be awakened (e.g., when one resource becomes available) to continue its execution.*

3. 📣 **notifyAll():**

The notifyAll() method wakes up all threads that are waiting on the object's monitor.

> 🙈 *Use notifyAll() when a change in the condition may be relevant to all waiting threads. For instance, when a producer adds an item to a queue that multiple consumers might be waiting for, you want to wake all waiting threads so they can re-check the condition.*

⚠️ Important:

These methods must be called from within a synchronized context (a synchronized block or method) on the same object whose monitor the thread is waiting on. They work together with a shared condition (often a flag or another shared variable) that threads check in a loop to handle spurious wakeups. 🔄🔍

Example:

Java

```java
public class WaitNotifyDemo {
    // Lock object used for synchronization
    private final Object lock = new Object();
    // Condition flag that threads check to decide whether to continue
    private boolean conditionMet = false;
    // Method where threads wait until conditionMet is true.
    public void doWait() {
        synchronized (lock) {
            while (!conditionMet) {  // Loop to avoid spurious wakeups
                try {
```

```
                    System.out.println(Thread.currentThread().getName() +
                    lock.wait();
                } catch (InterruptedException e) {
                    Thread.currentThread().interrupt();
                    System.out.println(Thread.currentThread().getName() +
                }
            }
            System.out.println(Thread.currentThread().getName() + " resume
        }
    }

    // Sets the condition to true and calls notify() so that one waiting t
    public void doNotify() {
        synchronized (lock) {
            conditionMet = true;
            System.out.println(Thread.currentThread().getName() + " called
            lock.notify();  // Wakes up one waiting thread (if any)
        }
    }

    // Sets the condition to true and calls notifyAll() so that all waitin
    public void doNotifyAll() {
        synchronized (lock) {
            conditionMet = true;
            System.out.println(Thread.currentThread().getName() + " called
            lock.notifyAll();  // Wakes up all waiting threads
        }
    }

    public static void main(String[] args) {
        // ****************************
        // Demonstrating notifyAll()
        // ****************************
        System.out.println("Demonstrating notifyAll():");
        WaitNotifyDemo demoAll = new WaitNotifyDemo();
        Thread waiter1 = new Thread(() -> demoAll.doWait(), "Waiter-1");
        Thread waiter2 = new Thread(() -> demoAll.doWait(), "Waiter-2");
        Thread waiter3 = new Thread(() -> demoAll.doWait(), "Waiter-3");
        waiter1.start();
        waiter2.start();
        waiter3.start();
```

```
        // Sleep to ensure all waiting threads have started and are waitin
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
        Thread notifierAll = new Thread(() -> demoAll.doNotifyAll(), "Noti
        notifierAll.start();
        try {
            waiter1.join();
            waiter2.join();
            waiter3.join();
            notifierAll.join();
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }

        // ****************************
        // Demonstrating notify()
        // ****************************
        System.out.println("nDemonstrating notify():");
        WaitNotifyDemo demoNotify = new WaitNotifyDemo();
        Thread waiterN1 = new Thread(() -> demoNotify.doWait(), "Waiter-N1
        Thread waiterN2 = new Thread(() -> demoNotify.doWait(), "Waiter-N2
        waiterN1.start();
        waiterN2.start();
        // Sleep to ensure waiting threads are waiting.
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
        Thread notifier = new Thread(() -> demoNotify.doNotify(), "Notifie
        notifier.start();
        // After calling notify(), only one waiting thread will resume whi
        // To ensure the program finishes, we call notifyAll() later to wa
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
```

```
        System.out.println("Calling notifyAll() to wake the remaining wait
        Thread notifier2 = new Thread(() -> demoNotify.doNotifyAll(), "Not
        notifier2.start();
        try {
            waiterN1.join();
            waiterN2.join();
            notifier.join();
            notifier2.join();
        } catch (InterruptedException e) {
            Thread.currentThread().interrupt();
        }
        System.out.println("Main thread: Execution finished.");
    }
}
```

Output : Because the order of thread execution is non-deterministic, the actual output may vary. A typical output might look like:

```
Demonstrating notifyAll():
Waiter-1 is waiting.
Waiter-2 is waiting.
Waiter-3 is waiting.
Notifier-All called notifyAll().
Waiter-2 resumed execution.
Waiter-1 resumed execution.
Waiter-3 resumed execution.

Demonstrating notify():
Waiter-N1 is waiting.
Waiter-N2 is waiting.
Notifier called notify().
Waiter-N1 resumed execution.
Calling notifyAll() to wake the remaining waiting thread.
Notifier2 called notifyAll().
Waiter-N2 resumed execution.
Main thread: Execution finished.
```

- 🕒 wait() releases the monitor and suspends the thread until notified.
- 🔔 notify() wakes up one waiting thread.
- 📣 notifyAll() wakes up all waiting threads.
- 🔄 These methods are used in coordinated thread communication (e.g., in producer–consumer scenarios) to signal condition changes and allow threads to resume execution in a

controlled manner.

Interview Questions:

1. Can you explain the producer-consumer problem and how to solve it using thread communication? 🧵❗
The producer-consumer problemis a classic example of inter-thread communication and synchronization in Java. It involves a producer that generates data (or items) and a consumer that processes those items.

A shared bounded buffer (or queue) is used to store the items. The challenge is to coordinate the producer and consumer so that:
• The producer waits when the buffer is full (to avoid overfilling).
• The consumer waits when the buffer is empty (to avoid consuming a non-existent item).

Inter-thread communication methods—wait(), notify(), and notifyAll()—are used to achieve this coordination. ✅

Java

```java
import java.util.LinkedList;
import java.util.Queue;
public class ProducerConsumer {
  // Shared buffer and its capacity
  private final Queue<Integer> buffer = new LinkedList<>();
  private final int CAPACITY = 5;

  // Method for the producer thread that adds items to the buffer.
  public void produce() throws InterruptedException {
    int value = 0;
    while (true) {
      synchronized (this) {
        // Wait while the buffer is full.
        while (buffer.size() == CAPACITY) {
          System.out.println("Buffer is full. Producer is waiting...");
          wait();
        }
        // Once there is space, produce an item.
        System.out.println("Producer produced: " + value);
        buffer.offer(value++);
        // Notify all waiting threads (consumers) that a new item is avail
```

```
        notifyAll();
      }
      // Sleep for a short time to simulate production time.
      Thread.sleep(1000);
    }
  }

  // Method for the consumer thread that takes items from the buffer.
  public void consume() throws InterruptedException {
    while (true) {
      synchronized (this) {
        // Wait while the buffer is empty.
        while (buffer.isEmpty()) {
          System.out.println("Buffer is empty. Consumer is waiting...");
          wait();
        }
        // Once there is an item, consume it.
        int value = buffer.poll();
        System.out.println("Consumer consumed: " + value);
        // Notify all waiting threads (producers) that space is available.
        notifyAll();
      }
      // Sleep for a short time to simulate consumption time.
      Thread.sleep(1500);
    }
  }

  // Main method to run the producer-consumer example.
  public static void main(String[] args) {
    ProducerConsumer pc = new ProducerConsumer();
    // Creating the producer thread.
    Thread producerThread = new Thread(new Runnable() {
      public void run() {
        try {
          pc.produce();
        } catch (InterruptedException e) {
          Thread.currentThread().interrupt();
          System.err.println("Producer thread interrupted.");
        }
      }
    }, "ProducerThread");
```

```
63
64      // Creating the consumer thread.
65      Thread consumerThread = new Thread(new Runnable() {
66        public void run() {
67          try {
68            pc.consume();
69          } catch (InterruptedException e) {
70            Thread.currentThread().interrupt();
71            System.err.println("Consumer thread interrupted.");
72          }
73        }
74      }, "ConsumerThread");
75
76      // Start both threads.
77      producerThread.start();
78      consumerThread.start();
79    }
80  }
```

Output :

```
Since thread scheduling is non-deterministic, the output may vary with each run. A
Producer produced: 0
Consumer consumed: 0
Producer produced: 1
Producer produced: 2
Consumer consumed: 1
Producer produced: 3
Producer produced: 4
Producer produced: 5
Buffer is full. Producer is waiting...
Consumer consumed: 2
Producer produced: 6
Consumer consumed: 3
Consumer consumed: 4
Producer produced: 7
…
```

In this output:

- 🏭 The producer prints a message when it produces an item.
- 🛑 If the buffer is full, the producer indicates that it is waiting.

• 🧑💻 The consumer prints a message when it consumes an item.
• ⏳ If the buffer is empty, the consumer indicates that it is waiting.
• 🔔 The calls to notifyAll() ensure that as soon as a change happens (an item is produced or consumed), the waiting threads are notified and re-check their conditions.

🧠 Explanation of the Code :

1. 📦 Shared Buffer and Capacity:
• We use a Queue<Integer> (implemented via a LinkedList) to serve as the buffer.
• A constant CAPACITY limits the number of items in the buffer.

2. 🔨 **Producer (produce() Method):**
• The producer enters a loop to continuously produce items.
• Inside a synchronized block (locking on the ProducerConsumer object), it checks if the buffer is full.
• If the buffer is full, the producer calls wait(), releasing the lock and suspending execution until notified.
• Once there's room in the buffer, the producer adds a new integer value to the buffer, then calls notifyAll() to wake any waiting consumers.
• A short sleep simulates production time.

3. 🧹 **Consumer (consume() Method):**
• The consumer similarly loops to continuously consume items.
• Inside a synchronized block, it waits while the buffer is empty, calling wait() to suspend execution.
• When an item is available, it retrieves (consumes) the item from the buffer and calls notifyAll() to signal the producer that space is now available.
• A sleep simulates consumption time.

4. 🔄 **Thread Communication:**
• wait() and notifyAll() are both used inside synchronized blocks to ensure proper coordination between threads.
• notifyAll() is used so that all waiting threads (be they producers or consumers) get a chance to re-check their condition and continue if possible.

5. 🚀 **Main Method:**
• Two threads are created—one for the producer and one for the consumer.
• They are started concurrently. As the threads run, you will see messages indicating when the producer or consumer is waiting, producing, or consuming items.

2. How does thread interruption work with communication methods? 🧵❗
Answer: When a thread is waiting (using wait(), join(), or blocking queue methods), it can be interrupted by another thread calling its interrupt() method. This causes an

InterruptedException to be thrown, allowing the waiting thread to handle the interruption. Proper handling involves either re-interrupting the thread or propagating the exception.

Java

```java
class ThreadInterruption {
  public static void main(String[] args) {
    Thread thread = new Thread(() -> {
      try {
        System.out.println("Thread: Going to sleep...");
        Thread.sleep(5000); // Sleep for 5 seconds
        System.out.println("Thread: Woke up!");
      } catch (InterruptedException e) {
        System.out.println("Thread: Interrupted!");
      }
    });
    thread.start();
    try {
      Thread.sleep(1000); // Main thread sleeps for 1 second
      thread.interrupt(); // Interrupt the sleeping thread
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }
}
```

Output :

```
Thread: Going to sleep...
Thread: Interrupted!
```

Conclusion 🎯

Effective thread communication is crucial for building concurrent applications that are both correct and performant. By using the appropriate communication mechanisms—whether low-level primitives like wait()/notify() or higher-level utilities from the java.util.concurrent package—developers can coordinate thread execution, share data safely, and avoid concurrency issues like race conditions and deadlocks. 👨‍💻💪

Understanding thread communication patterns enables you to design robust multithreaded systems that can take full advantage of modern multicore processors while maintaining data integrity and application responsiveness. 🚀📊