JAVA MULTITHREADING: THREAD INTERRUPTION, FORK/JOIN FRAMEWORK, AND DEADLOCK PREVENTION

Sandra Kumi

REVIEWER Mr. Thomas Darko

Java Multithreading: Thread Interruption, Fork/Join Framework, and Deadlock Prevention

---

1. Thread Interruption

Thread interruption is a mechanism in Java to signal a thread that it should stop its current operation. It does not forcefully stop the thread; instead, the thread is requested to self-terminate by checking its interrupted status.

Key Methods

- Thread.interrupt(): Signals the thread to interrupt.
- Thread.isInterrupted(): Checks if the thread has been interrupted.
- InterruptedException: Thrown when a thread is blocked in operations like sleep() or wait() and gets interrupted.

Example:

```
class InterruptTask implements Runnable {

    @Override

    public void run() {

        try {
            for (int i = 1; i <= 5; i++) {
                System.out.println(Thread.currentThread().getName() + " performing task " + i);
                Thread.sleep(1000);
            }
        } catch (InterruptedException e) {
            System.out.println(Thread.currentThread().getName() + " was interrupted.");
        }
    }
}


public class ThreadInterruptionExample {
    public static void main(String[] args) {
        Thread workerThread = new Thread(new InterruptTask(), "WorkerThread");


        workerThread.start();
        try { Thread.sleep(3000); } catch (InterruptedException e) { }

        workerThread.interrupt();  // Request interruption

        System.out.println("Main thread requested interruption.");

    }

}
```

In this example, the WorkerThread is interrupted during its sleep, which causes the thread to exit gracefully by catching the InterruptedException.

---

2. Fork/Join Framework

The Fork/Join framework is part of the Java java.util.concurrent package and is designed to handle recursive divide-and-conquer tasks. It leverages multiple threads to divide the task into smaller subtasks that can be processed in parallel, improving efficiency for tasks like sorting or summing large datasets.

Key Components

- ForkJoinPool: Manages worker threads.
- RecursiveTask: Represents a task that returns a result.
- RecursiveAction: Represents a task that does not return a result.

Example:

```
import java.util.concurrent.RecursiveTask;

import java.util.concurrent.ForkJoinPool;



class SumTask extends RecursiveTask<Integer> {

    private static final int THRESHOLD = 10;

    private int[] numbers;

    private int start;

    private int end;



    public SumTask(int[] numbers, int start, int end) {

        this.numbers = numbers;

        this.start = start;

        this.end = end;

    }
```

```
@Override
protected Integer compute() {
    int length = end - start;
    if (length <= THRESHOLD) {
        int sum = 0;
        for (int i = start; i < end; i++) {
            sum += numbers[i];
        }
        return sum;
    } else {
        int mid = start + length / 2;
        SumTask leftTask = new SumTask(numbers, start, mid);
        SumTask rightTask = new SumTask(numbers, mid, end);
```

```
            leftTask.fork();  // Execute left task in parallel

            int rightResult = rightTask.compute();  // Right task in current thread

            int leftResult = leftTask.join();  // Wait for left task



            return leftResult + rightResult;

        }

    }

}



public class ForkJoinExample {

    public static void main(String[] args) {

        int[] numbers = new int[100];

        for (int i = 0; i < 100; i++) {

            numbers[i] = i + 1;

        }
        ForkJoinPool pool = new ForkJoinPool();

        SumTask task = new SumTask(numbers, 0, numbers.length);



        int sum = pool.invoke(task);

        System.out.println("Sum: " + sum);

    }

}
```

In this example, the SumTask recursively divides an array of numbers and sums them in parallel using the Fork/Join framework. The task is split into smaller chunks, with each chunk being processed by different threads.

---

3. Deadlock and Deadlock Prevention

A deadlock occurs when two or more threads are blocked forever, waiting for each other to release resources. In Java, this typically happens when two threads hold locks on different objects and each thread is trying to acquire a lock that the other holds.

Deadlock Scenario:

```
class Resource {
    public synchronized void useResource(Resource other) {

        System.out.println(Thread.currentThread().getName() + " is using resource.");

        other.release();

    }



    public synchronized void release() {

        System.out.println(Thread.currentThread().getName() + " is releasing resource.");

    }

}



public class DeadlockExample {

    public static void main(String[] args) {

        Resource resource1 = new Resource();

        Resource resource2 = new Resource();
    Thread t1 = new Thread(() -> {

      synchronized (resource1) {

        System.out.println(Thread.currentThread().getName() + " locked Resource 1.");

        try { Thread.sleep(100); } catch (InterruptedException e) {} synchronized (resource2) {
System.out.println(Thread.currentThread().getName() + " locked Resource 2.");
resource1.useResource(resource2); } } }, "Thread 1");

  Thread t2 = new Thread(() -> {

    synchronized (resource2) {

      System.out.println(Thread.currentThread().getName() + " locked Resource 2.");

      try { Thread.sleep(100); } catch (InterruptedException e) {}

      synchronized (resource1) {

        System.out.println(Thread.currentThread().getName() + " locked Resource 1.");

        resource2.useResource(resource1);

      }

    }

  }, "Thread 2");
```

```
    t1.start();

    t2.start();

}

}
```

In this deadlock scenario, `Thread 1` locks `Resource 1` and waits for `Resource 2`, while `Thread 2` locks `Resource 2` and waits for `Resource 1`, resulting in a deadlock.

Deadlock Prevention:

To avoid deadlock, it's essential to ensure that all threads acquire locks in the same order. This can be done by imposing a strict ordering on the resources.

Solution: Lock Ordering

```
public class DeadlockSolution {

    public static void main(String[] args) {
```

```
Resource resource1 = new Resource();

Resource resource2 = new Resource();



Thread t1 = new Thread(() -> {

    synchronized (resource1) {

        synchronized (resource2) {

            resource1.useResource(resource2);

        }

    }

}, "Thread 1");



Thread t2 = new Thread(() -> {

    synchronized (resource1) {  // Lock order fixed

        synchronized (resource2) {

            resource2.useResource(resource1);
```

```
                }
            }
        }, "Thread 2");

        t1.start();
        t2.start();
    }
}
```

In this solution, both threads acquire locks in the same order (first resource1, then resource2), which prevents deadlock.

Conclusion

Understanding and managing threads in Java is crucial for developing efficient, multithreaded applications. This document has covered key concepts such as thread interruption, the Fork/Join framework for parallel task execution, and strategies for preventing deadlocks. By applying these techniques, you can ensure safe and efficient thread management in complex applications.