Ordered Printing

This problem is about imposing an order on thread execution.

We'll cover the following

- Problem Statement
- Solution
- Solution using CountDownLatch

Problem Statement#

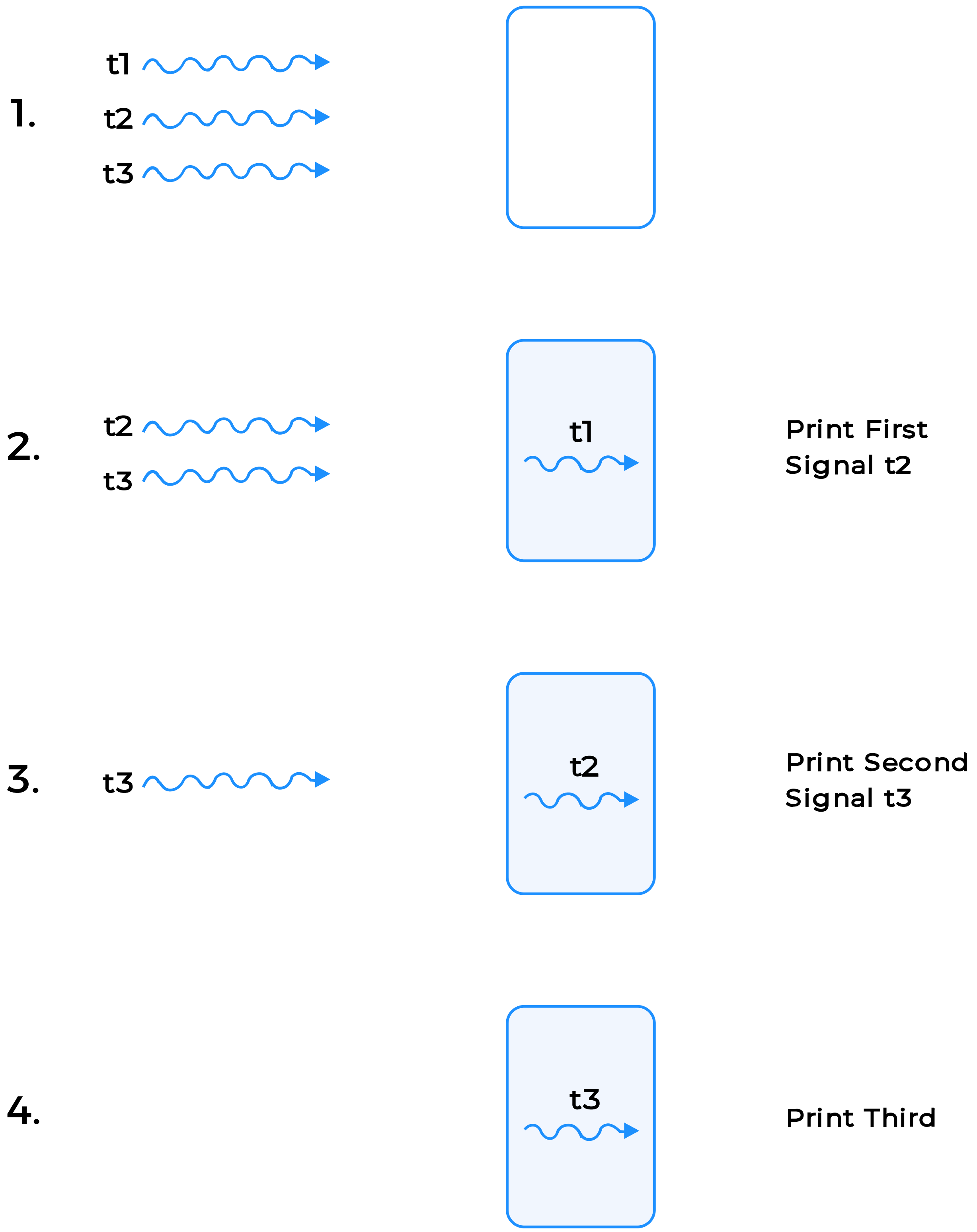
Suppose there are three threads t1, t2 and t3. t1 prints **First**, t2 prints **Second** and t3 prints **Third**. The code for the class is as follows:

```
public class OrderedPrinting {

    public void printFirst() {
       System.out.print("First");
    }

    public void printSecond() {
       System.out.print("Second");
    }

    public void printThird() {
       System.out.print("Third");
    }

}
```

Thread t1 calls printFirst(), thread t2 calls printSecond(), and thread t3 calls printThird(). The threads can run in any order. You have to synchronize the threads so that the functions **printFirst()**, **printSecond()** and **printThird()** are executed in order.

The workflow of the program is shown below:

Ordered Printing



1.

2.

Print First
Signal t2

3.

Print Second
Signal t3

4.

Print Third

Solution#

We present two solutions for this problem; one using the basic `wait()` & `notifyAll()` functions and the other using **CountDownLatch**.

Solution 1

In this solution, we have a class `OrderedPrinting` that consists of a private variable; `count`. The class consists of 3 functions `printFirst()`,`printSecond()` and `printThird()`. The structure of the class is as follows:

```
class OrderedPrinting {
    int count;

    public OrderedPrinting() {
        count = 1;
    }

    public void printFirst() {
    }

    public void printSecond() {
    }

    public void printThird() {
    }
}
```

In the constructor, `count` is initialized with 1. Next we will explain the printFirst() function below:

```
public void printFirst() throws InterruptedException {

    synchronized(this) {
        System.out.println("First");
        count++; //for printing Second, increment count
        this.notifyAll();
    }
    }
}
```

In `printFirst()`, "First" is printed. We do not need to check the value of `count` here. After printing, `count` is incremented for the next word to be printed. Any waiting threads are then notified via `notifyAll()`, signalling them to proceed.

```
public void printSecond() throws InterruptedException {

    synchronized(this) {
        while(count != 2) {
            this.wait();
        }
        System.out.println("Second");
        count++;
        this.notifyAll();
    }
}
```

In the second method, the value of `count` is checked. If it is not equal to 2, the calling thread goes into wait. When the value of `count` reaches 2, the while loop is broken and "Second" is printed. The value of `count` is incremented for the next number to be printed and `notifyAll()` is called.

```
public void printThird() throws InterruptedException {

    synchronized(this) {
        while(count != 3) {
            this.wait();
        }
        System.out.println("Third");
    }
}
```

The third method checks works in the same way as the second. The only difference being the check for `count` to be equal to 3. If it is, then "Third" is printed otherwise the calling thread waits.

To run our proposed solution, we will create another class to achieve multi-threading. When we extend `Thread` class, each of our thread creates a unique object and associates with the parent class. This class has two variables: one is the object of `OrderedPrinting` and the other is a string variable `method`. The string parameter checks the method to be invoked from `OrderedPrinting`.

```
class OrderedPrintingThread extends Thread {
    private OrderedPrinting obj;
    private String method;

    public OrderedPrintingThread(OrderedPrinting obj, String method) {
        this.method = method;
        this.obj = obj;
    }

    public void run() {
        //for printing "First"
        if ("first".equals(method)) {
            try {
                obj.printFirst();
            }
            catch(InterruptedException e) {

            }
        }
        //for printing "Second"
        else if ("second".equals(method)) {
            try {
                obj.printSecond();
            }
            catch(InterruptedException e) {

            }
        }
        //for printing "Third"
        else if ("third".equals(method)) {
            try {
                obj.printThird();
            }
            catch(InterruptedException e) {

            }
        }
    }
}
```

We will be creating 3 threads in the `Main` class for testing each solution. Each thread will be passed the same object of `OrderedPrinting`. **t1** will call `printFirst()`, **t2** will call `printSecond()` and **t3** will call `printThird()`. The output shows printing done in the proper order i.e first, second and third irrespective of the calling order of threads.

```java
class OrderedPrinting {

    int count;

    public OrderedPrinting() {
        count = 1;
    }

    public void printFirst() throws InterruptedException {

        synchronized(this){
            System.out.println("First");
            count++;
            this.notifyAll();
        }
    }

    public void printSecond() throws InterruptedException {

        synchronized(this){
            while(count != 2){
                this.wait();
            }
            System.out.println("Second");
            count++;
            this.notifyAll();
        }

```

Solution using `CountDownLatch`#

The second solution includes the use of **CountDownLatch**; a synchronization utility used to achieve concurrency. It manages multithreading where a certain sequence of operations or tasks is required. Everytime a thread finishes its work, `countdown()` is invoked, decrementing the counter by 1. Once this count reaches zero, `await()` is notified and control is given back to the main thread that has been waiting for others to finish.

The basic structure of the class `OrderedPrinting` is the same as presented in solution 1 with the only difference of using **countdownlatch** instead of **volatile** variable. We have 2 **countdownlatch** variables that get initialized with 1 each.

```
class OrderedPrinting {
    CountDownLatch latch1;
    CountDownLatch latch2;

    public OrderedPrinting() {
        latch1 = new CountDownLatch(1);
        latch2 = new CountDownLatch(1);
    }
}
```

In `printFirst()` method, `latch1` decrements and reaches 0, waking up the waiting threads consequently. In `printSecond()`, if `latch1` is free (reached 0), then the printing is done and `latch2` is decremented. Similarly in the third method `printThird()`, `latch2` is checked and printing is done. The latches here act like switches/gates that get closed and opened for particular actions to pass.

```
public void printFirst() throws InterruptedException {
    //print and notify waiting threads
    System.out.println("First");
    latch1.countDown();
}
```

```
public void printSecond() throws InterruptedException {
    //wait if "First" has not been printed yet
    latch1.await();
    //print and notify waiting threads
    System.out.println("Second");
    latch2.countDown();
}
```

```
public void printThird() throws InterruptedException {
    //wait if "Second" has not been printed yet
    latch2.await();
    System.out.println("Third");
}
```

As in the previous solution, we create `OrderedPrintingThread` class which extends the `Thread` class. Details of this class are explained at length above.

```
import java.util.concurrent.CountDownLatch;

class OrderedPrinting
{
    CountDownLatch latch1;
    CountDownLatch latch2;

    public OrderedPrinting()
    {
        latch1 = new CountDownLatch(1);
        latch2 = new CountDownLatch(1);
    }

    public void printFirst() throws InterruptedException
    {
        System.out.println("First");
        latch1.countDown();
    }

    public void printSecond() throws InterruptedException
    {
        latch1.await();
        System.out.println("Second");
        latch2.countDown();
    }

    public void printThird() throws InterruptedException
    {
```

← Back

Epilogue

Next →

Printing Foo Bar n Times

☑ Mark as Completed

Report an Issue

Ask a Question
(https://discuss.educative.io/tag/ordered-printing__bonus-questions__java-multithreading-for-senior-engineering-interviews)