Printing Number Series (Zero, Even, Odd)

This problem is about repeatedly executing threads which print a specific type of number. Another variation of this problem; print even and odd numbers; utilizes two threads instead of three.

We'll cover the following

Problem Statement#

Suppose we are given a number *n* based on which a program creates the series *010203...0n*. There are three threads t1, t2 and t3 which print a specific type of number from the series. t1 only prints zeros, t2 prints odd numbers and t3 prints even numbers from the series. The code for the class is given as follows:

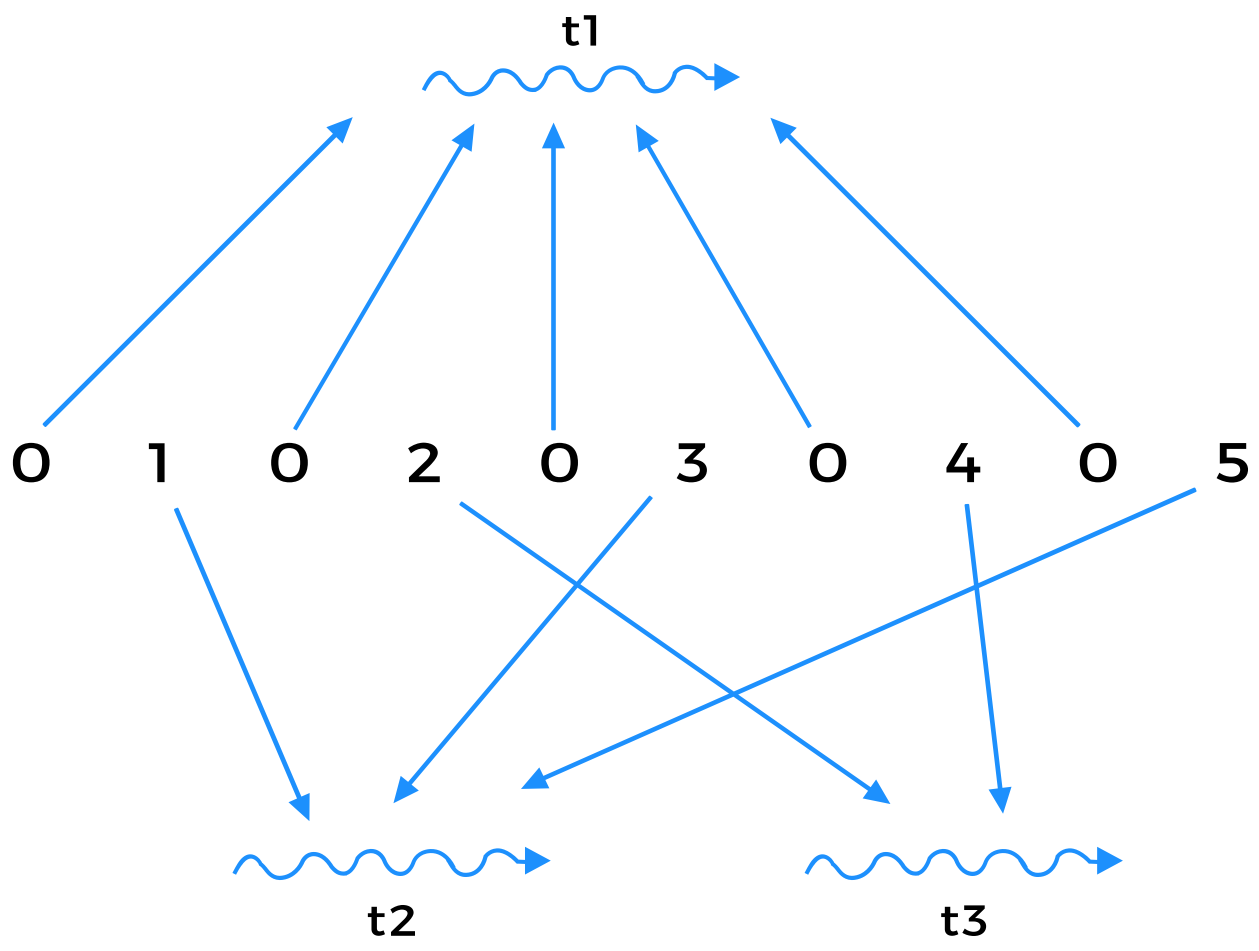
```
class PrintNumberSeries {

    public PrintNumberSeries(int n) {
        this.n = n;
    }

    public void PrintZero() {
    }
    public void PrintOdd() {
    }
    public void PrintEven() {
    }
}
```

You are required to write a program which takes a user input n and outputs the number series using three threads. The three threads work together to print zero, even and odd numbers. The threads should be synchronized so that the functions PrintZero(), PrintOdd() and PrintEven() are executed in an order.

The workflow of the program is shown below:



Solution#

This problem is solved by using Semaphores in Java. Semaphores are used to restrict the number of threads that can access some (physical or logical) resource at the same time. Our solution makes use of three semaphores;

zeroSem for printing zeros, oddSem for printing odd numbers and evenSem for printing even numbers. The basic structure of the class is given below:

```
class PrintNumberSeries {

    private int n;
    private Semaphore zeroSem, oddSem, evenSem;

    public PrintNumberSeries(int n) {
    }

    public void PrintZero() {
    }

    public void PrintOdd() {
    }

    public void PrintEven() {
    }

}
```

`n` is the user input that prints the series till *n*th number. The constructor of this class appears below:

```
public PrintNumberSeries(int n) {
        this.n = n;
        zeroSem = new Semaphore(1);
        oddSem = new Semaphore(0);
        evenSem = new Semaphore(0);
}
```

The argument passed to semaphore's constructor is the number of 'permits' available. For `oddSem` and `evenSem`, all `acquire()` calls will be blocked initially as they are initialized with 0. For `zeroSem`, the first `acquire()` call will succeed as it is intialized with 1. The code of the first method `PrintZero()` is as follows:

```
1. public void PrintZero() {
2.    for (int i = 0; i < n; ++i) {
3.        zeroSem.acquire();
4.        System.out.print("0");
5.        // release oddSem if i is even else release evenSem i
f i is odd
6.        (i % 2 == 0 ? oddSem : evenSem).release();
7.    }
8.}
```

`PrintZero()` begins with a loop iterating from 0 till `n` (exclusive). The semaphore `zeroSem` is acquired and '0' is printed. A very significant line in this method is **line 6** in the loop. The modulus operator **(%)** gives the remainder of a division by the value following it. In our case, the current value is divided by 2 to determine if `i` is even or odd. If `i` is odd, then it means we just printed an odd number and the next number in the sequence will be an even number so, `evenSem` is released. In the same way if `i` is even then `oddSem` is released for printing the next odd number in the sequence. The second method `PrintOdd()` is shown below:

```
public void PrintOdd() {
    for (int i = 1; i <= n; i += 2) {
        oddSem.acquire();
        System.out.print(i);
        zeroSem.release();
    }
}
```

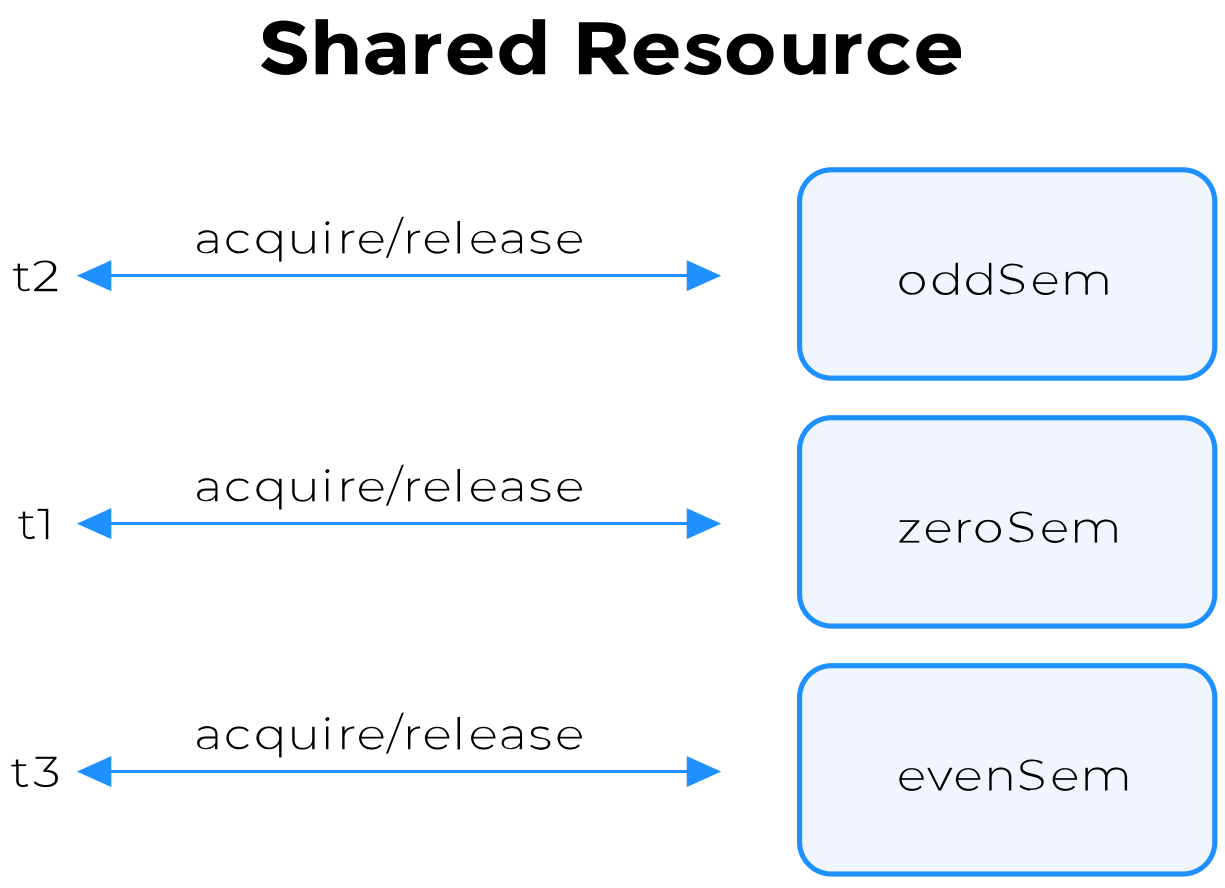
The loop iterates from 1 till `n` (inclusive) and `i` is incremented by 2 after each iteration to ensure that only odd numbers are printed. `oddSem` is acquired when `PrintZero()` releases it after determining that it is the turn for an odd number to be printed. Since zero is required to be printed before every even or odd number, `zeroSem` is released after printing the odd number.

The last method of the class `PrintEven()` is shown below:

```
public void PrintEven() {
    for (int i = 2; i <= n; i += 2) {
        evenSem.acquire();
        System.out.print(i);
        zeroSem.release();
    }
}
```

`PrintEven()` operates in the same manner as `PrintOdd()` except that its loop begins from 2. `evenSem` is acquired if it is released by `PrintZero()` and an even number is printed. `zeroSem` is released for zero to be printed next. If `n` is reached, the loop breaks.

The working of this class can be seen as a lock-shift phenomenon where every method is given the control at its turn and blocked otherwise. `n` is manipulated by only one thread at a time.



To test our solution, We will create 3 threads `t1`, `t2` and `t3` in `Main` class. `t1` prints 0, `t2` prints odd numbers and `t3` prints even numbers. The threads are started in random order.

```
import java.util.concurrent.*;

class PrintNumberSeries {
    private int n;
    private Semaphore zeroSem, oddSem, evenSem;

    public PrintNumberSeries(int n) {
        this.n = n;
        zeroSem = new Semaphore(1);
        oddSem = new Semaphore(0);
        evenSem = new Semaphore(0);
    }

    public void PrintZero() {
        for (int i = 0; i < n; ++i) {
            try {
                zeroSem.acquire();
            }
            catch (Exception e) {
            }
            System.out.print("0");
            // release oddSem if i is even or else release evenSem if i is odd
            (i % 2 == 0 ? oddSem : evenSem).release();
        }
    }

    public void PrintEven() {
        for (int i = 2; i <= n; i += 2) {
            try {
                evenSem.acquire();
            }
            catch (Exception e) {
            }
            System.out.print(i);
            zeroSem.release();
        }
    }

    public void PrintOdd() {
        for (int i = 1; i <= n; i += 2) {
            try {
                oddSem.acquire();
            }
            catch (Exception e) {
            }
            System.out.print(i);
            zeroSem.release();
        }
    }
}
```

```
class PrintNumberSeriesThread extends Thread {


    PrintNumberSeries zeo;
    String method;

    public PrintNumberSeriesThread(PrintNumberSeries zeo, String method){
        this.zeo = zeo;
        this.method = method;
    }

    public void run() {
        if ("zero".equals(method)) {
            try {
                zeo.PrintZero();
            }
            catch (Exception e) {
            }
        }
        else if ("even".equals(method)) {
            try {
                zeo.PrintEven();
            }
            catch (Exception e) {
            }
        }
        else if ("odd".equals(method)) {
            try {
                zeo.PrintOdd();
            }
            catch (Exception e) {
            }
        }
    }
}

public class Main {

    public static void main(String[] args) {

            PrintNumberSeries zeo = new PrintNumberSeries(5);

            Thread t1 = new PrintNumberSeriesThread(zeo,"zero");
            Thread t2 = new PrintNumberSeriesThread(zeo,"even");
            Thread t3 = new PrintNumberSeriesThread(zeo,"odd");

            t2.start();
            t1.start();
            t3.start();

    }
}
```

We were told to use three threads in the problem statement but the solution can be achieved using two threads as well. Since zero is printed before every number, we do not need to dedicate a special thread for it. We can simply print a zero before printing every odd or even number.

Back
Printing Foo Bar n Times

Next
Build a Molecule

Mark as Completed

Report an Issue

Ask a Question
(https://discuss.educative.io/tag/printing-number-series-zero-even-odd__bonus-questions__java-multithreading-for-senior-engineering-interviews)