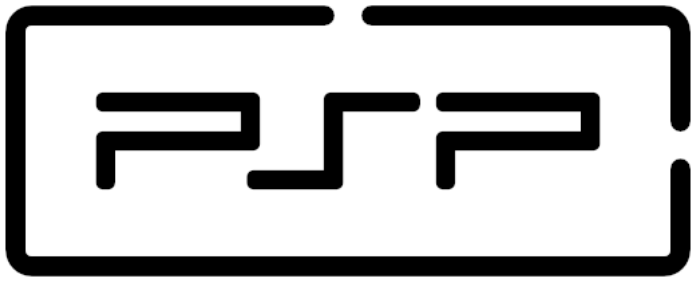


Process and Service Programming

3.3 Producer-Consumer model

3.3 Producer-Consumer model

- 3.3.1. Communication & synchronization template
- 3.3.2 Main class
- 3.3.3 Producer & Consumer classes
- 3.3.4 Shared class. Threads synchronization

3.3.1. Communication & synchronization template

Threads synchronization means having tools to avoid `starvation` (threads lock), `deadlocks` (when a condition can never be satisfied) and to ensure shared resources are well managed by concurrent threads access.

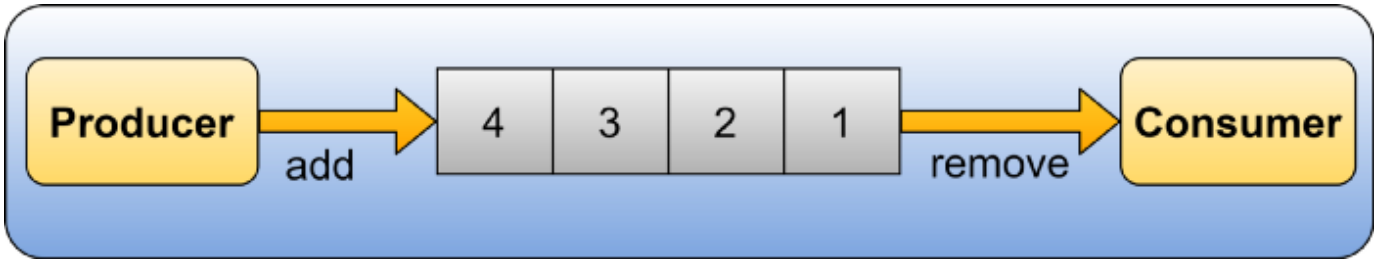
The Producer-Consumer problem is a classic example of a multi-threaded synchronization problem. Let's go into the usage of the shared resources by using this famous algorithm. Wikipedia.

Without thread control mechanism we already know that problems will rise up randomly:

- Consumer can get elements more than once, exceeding the stock (banc account balance under 0, reader reading a book before it is finished).
- Producers can be quicker than Consumer and produce more information than the system can get, making data loose.- Consumer can be quicker than the Producer and can get more than once the same value, having inconsistent systems.

That's all we know as `race conditions`.

The following code template repeats over and over again for almos all activities we are going to work on. That's what we call the Producer-Consumer model.



This model is based on three classes, but depending on the problem we can have only producers or just consumers.

Model as a design pattern

It's very important to fit our code into the model schema

This is like a puzzle where we have to adjust the problem solution. Sometimes we won't have a producer, other there will be no consumer. Maybe we will use the wait condition only in one of them.

We shouldn't add or modify the way the schema is presented, all parts must fit into the given model.

3.3.2 Main class

> Main class will always have the same estructure. Following code can be used as a a template.

Here we instantiate the shared object to be used by producers&consumers. This is the object that will hold communication, synchronization and information exchange between threads.

In this example it is an object, but we can use a Collection or any other data structure useful for thread to share information and synchronize.

```java
public class ClasePrincipal {

    public static void main(String[] args) {
        ClaseCompartida objetoCompartido = new ObjetoCompartido();
        Productor p  = new Productor(objetoCompartido);
        Consumidor c  = new Consumidor(objetoCompartido);
        productor.start();
        consumidor.start();

        // No es obligatorio, pero en ocasiones puede interesar que la ClasePrincipal
        // espere a que acaben los hilos
        productor.join();
        consumidor.join();

        // Acciones a realizar una vez hayan acabado el productor y el consumidor
    }

}
```

Number of producer & consumer threads

In the previous code we have created and launched one of each, but it has not to be like that.

Each problem to solve will need a different number of *Producers* and *Consumers*, that will be instantiated and launched in the main method or in any other complementary method in the class in charge of thread management.

In the same way, it's on the problem if the main thread has to wait for the others to finish or not.

3.3.3 Producer & Consumer classes

> Both **Producer** and **Consumer** classes will call methods in the shared object.

In both classes, the application logic will be developed inside **run** method. This will be done basically accessing the shared object, calling its synchronized methods, modifying its properties and updating the object state to control its functionality.

```java
public class Consumidor extends Thread {
    private ClaseCompartida objetoCompartido;

    Consumidor(ClaseCompartida objetoCompartido) {
        this.objetoCompartido = objetoCompartido;
    }

    @Override
    public void run() {
        // La ejecución del método run estará normalmente gestionada por un bucle
        // que controlará el ciclo de vida del hilo y se adaptará al problema.
        // En el caso de simulaciones se harán esperas proporcionales.
        try {
            // Código que hace el hilo consumidor
            objetoCompartido.accionDeConsumir();
            // La espera es opcional
            Thread.sleep((long)(Math.random()*1000+1000));
        } catch (InterruptedException ex) {

        }
    }
}
```

```java
public class Productor extends Thread {
    private ClaseCompartida objetoCompartido;

    Productor(ClaseCompartida objetoCompartido) {
        this.objetoCompartido = objetoCompartido;
    }

    @Override
    public void run() {
        // La ejecución del método run estará normalmente gestionada por un bucle
        // que controlará el ciclo de vida del hilo y se adaptará al problema.
        // En el caso de simulaciones se harán esperas proporcionales.
        try {
            // Código que hace el hilo productor
            objetoCompartido.accionDeProducir();
            // La espera es opcional
            Thread.sleep((long)(Math.random()*1000+1000));
        } catch (InterruptedException ex) {

        }
    }
}
```

3.3.4 Shared class. Threads synchronization

This model is completed with the shared object class. Here we provide methods to be used by both Producers and Consumers. Furthermore, this class must be thread-safe to avoid `race conditions`.

```java
class ClaseCompartida {
    int valorAccedidoSimultaneamente;

    ClaseCompartida() {
        // Se inicializa el valor
        this.valorAccedidoSimultaneamente = 0;
    }

    public synchronized void accionDeConsumir() {
        // Si no se cumple la condición para poder consumir, el consumidor debe esperar
        while (valorAccedidoSimultaneamente() == 0) {
            try {
                System.out.println("Consumidor espera...");
                wait();
            } catch (InterruptedException ex) {
                // Si es necesario se realizará la gestión de la Interrupción
            }
        }

        // Cuando se ha cumplido la condición para consumir, el consumidor consume
        valorAccedidoSimultaneamente--;
        System.out.printf("Se ha consumido: %d.\n", valorAccedidoSimultaneamente);

        // Se activa a otros hilos que puedan estar en espera
        notifyAll();
    }

    public synchronized void accionDeProducir () {
        // Si no se cumple la condición para poder producir, el productor debe esperar
        while (valorAccedidoSimultaneamente() > 10) {
            try {
                System.out.println("Productor espera...");
                wait();
            } catch (InterruptedException ex) {
                // Si es necesario se realizará la gestión de la Interrupción
            }
        }

        // Cuando se ha cumplido la condición para producir, el productor produce
        valorAccedidoSimultaneamente++;
        System.out.printf("Se ha producido: %d.\n", valorAccedidoSimultaneamente);

        // Se activa a otros hilos que puedan estar en espera
        notifyAll();
    }
```

What's interesting from above code is the pair `wait / notifyAll` together with the `synchronized` modifier.

- A call to a **synchronized** method makes it be run if and only if there is no other thread running another **synchronized** method `for the same object instance`. If that happens the thread trying to access the synchronized block will be locked until another thread leaves the synchronized block. Then one random thread is chosen from the threads waiting for the monitor and then it owns the monitor and runs the synchronized block.
- Simply put, calling **wait()** forces the current thread to wait until some other thread invokes **notify()** or **notifyAll()** on the same object. For this, the current thread must own the object's monitor, because the monitor will be released after the wait call.
- We use the **notify/notifyAll** methods for waking up threads that have previously made a wait() call for this monitos. All awaken threads are automatically sent onto the monitor queue together with all threads already waiting to own the monitor. All threads, once the monitor is owned by them, will start running the synchronized code o will continue running the next sentence after the wait call.

> With **wait**, **notifyAll** methods and **synchronized** code blocks we can avoid concurrent threads to modify a shared variable. *(lines 21 and 40 from previous code).*

Producer-Consumer model summary

Original Producer-Consumer works with a buffer where the Producer puts information and the Consumer gets it from the buffer. The buffer can never be overflown and it cannot be read if it is empty.

Our example has been simplified by using a int variable that has to be always in the range [0.10]

This variables can be of any type and the class code will be different depending on it. It must be valid for the problem and the data type control.

Finally, conditions or states added for waiting and updates will be what us, as programmers, must code in order to make it work as specified by problems requirements.