Descargar estos apuntes

Índice

Unidad 3. Programación multihilo: Modelo productor-consumidor

Esquema de sincronización y comunicación de hilos

La sincronización de threads implica disponer de mecanismos que permitan asegurar que no se producen situaciones de `inanición o starvation` (bloqueo de hilos como consecuencia del acceso a recursos compartidos limitados), `interbloqueos` (espera por parte de los hilos cuando una condición no puede ser satisfecha) y que, por lo tanto, se opera de forma correcta con los recursos compartidos por hilos concurrentes.

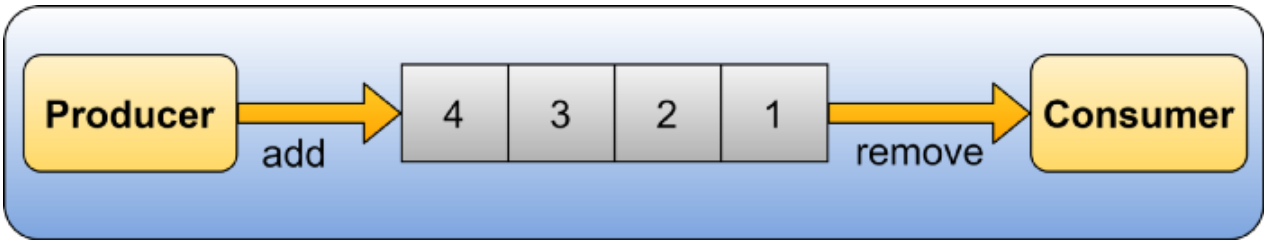
En esta sección vamos a ejemplificar la compartición de recursos a través de un objeto contenedor (objeto compartido) mediante el famoso algoritmo del Productor-Consumidor que podemos ver resumido en Wikipedia.

Si no se implementasen medidas de control, ya hemos visto que podrían darse situaciones anómalas como:

- El consumidor puede obtener los elementos producidos más de una vez, excediendo la producción del mismo (poder dejar la cuenta en números rojos en el ejemplo del banco, o que un lector pueda leer un libro antes de estar terminado).
- El productor sea más rápido que el Consumidor y genere más información de la que el sistema pueda almacenar, o bien parte de la información que genere se pierda sin que un Consumidor la haya recuperado.
- El Consumidor sea más rápido que el Productor y puede terminar consumiendo dos o más veces el mismo valor, generando información inconsistente en el sistema.

Todas estas circunstancias son las que conocemos como condiciones de carrera o `race conditions`.

El esquema de clases representado por este modelo se repite fielmente entre los diferentes ejercicios que vamos a realizar, es lo que denominamos el modelo Productor-Consumidor.



Este modelo se basa en tres clases, aunque dependiendo del problema, podemos encontrarnos que no tenemos **productor** o **consumidor**.

> **Modelo como patrón de diseño**
>
> Es importante que nos ajustemos al esquema presentado en el modelo.

> Como ya se ha dicho, a veces no habrá productor, otras no estará el consumidor, en otras el código de bloqueo estará sólo en una de las clases, pero no debemos inventar ni añadir nada al esquema, debemos encajar el problema a solucionar dentro del código proporcionado.

Clase Principal

> La clase principal siempre va a tener la misma estructura. El siguiente código se puede usar como plantilla

En esta clase se declara el objeto o propiedad que van a compartir el productor y el consumidor. Este objeto es a través del que se realiza la comunicación, sincronización e intercambio de información entre los hilos.

Aquí se representa como un objeto, aunque puede ser una Colección o cualquier estructura de datos que puedan compartir los hilos.

```
public class ClasePrincipal {

    public static void main(String[] args) {
        ClaseCompartida objetoCompartido = new ClaseCompartida();
        Productor productor  = new Productor(objetoCompartido);
        Consumidor consumidor  = 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
    }

}
```

> ⚠ **Número de hilos por tipo**
>
> En el ejemplo se crea un hilo de cada tipo. Esto no tiene porqué ser así.
>
> Cada problema determinará el número de hilos *Productores* y *Consumidores* necesarios, por lo que será en este método main, o en algún otro método de la *ClasePrincipal* donde se realice la gestión de los hilos.
>
> De igual forma, dependerá de cada problema si el hilo principal debe esperar a que el resto finalice o no.

Clase Productor y Consumidor

> Por otro lado vamos a tener la clase del **productor** y del **consumidor** que se encargan de realizar las llamadas necesarias a los métodos del objeto compartido que reciben como parámetro.

Estas dos clases son las que van a tener, dentro del método **run**, la lógica de la aplicación, accediendo al objeto compartido, modificando las propiedades compartidas entre los diferentes hilos (productores y/o consumidores) y actualizando el estado del objeto compartido para que module su funcionalidad.

```
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) {

        }
    }
}
```

Clase Compartida. Sincronización de hilos

El modelo se completa con la clase compartida. Aquí vamos a crear los métodos a los que acceden productores y consumidores y, además, vamos a realizar la sincronización entre hilos para que no se produzcan `condiciones de carrera`.

```
public 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();
    }
}
```

Lo interesante del código anterior, como ya hemos visto con anterioridad, es la pareja de métodos `wait / notifyAll`, junto con el modificador `synchronized`.

- Cuando se llama a un método **synchronized** este método se ejecuta sí y solo sí no hay otro hilo ejecutando otro método **synchronized** `del mismo objeto`. Si se diese ese caso, el hilo se quedará en espera hasta que otro hilo libere el monitor. En ese momento todos los hilos que estén esperando se despiertan y sólo uno de ellos, el que consigue el monitor, puede ejecutar el código **synchronized" en exclusión mutua.
- Cuando se hace una llamada al método **wait**, un hilo se queda en espera y, además, libera el bloqueo del monitor. El hilo se quedará en espera hasta que otro hilo ejecute una llamada de señalización (**notify/notifyAll**).
- Cuando se hace una llamada al método **notify** o **notifyAll**, uno o todos los hilos que están en espera por haber hecho **wait** se despiertan y pasan a esperar poder tomar el control del bloqueo del **synchronized**. A partir de ese momento, de forma aleatoria, uno de ellos o de los que ya estaban en la cola del bloqueo **synchronized** toma el control y o bien empieza o bien sigue ejecutándose por donde se quedó (en caso de que hubiese llamado a **wait**).

> Con los métodos **wait**, **notify/notifyAll** y los bloques de código **synchronized** se consigue evitar que varios hilos puedan modificar a la vez una variable *(Ver líneas 21 y 40 del ejemplo anterior)*.

Resumen del modelo Productor-Consumidor

El modelo Productor-Consumidor original trabaja con un buffer en el que el Productor va depositando información y el Consumidor la va sacando, de forma que el buffer nunca se llene ni se pueda leer si está vacío.

En nuestro ejemplo, lo hemos simplificado al uso de una variable que nunca puede exceder el valor de 10 ni ser inferior a 0.

Como ya hemos dicho, esa variable puede ser cualquier tipo de dato, y el código de las clases variará en función de ello, para adaptarlo al problema y al control del tipo de dato utilizado.

Además, las condiciones o estados que se utilizan para las esperas y las actualizaciones será lo que nosotros, como programadores, tengamos que adaptar al modelo para hacerlo funcionar en situaciones diferentes.