2. Programación concurrente

Anexo I. Procesos, prioridad de hilos, productores y consumidores

Programación de Servicios y Procesos

Arturo Bernal
Nacho Iborra
Javier Carrasco

Tabla de contenido

1. Gestión de procesos en Java

Ahora que ya sabes qué es un proceso, se verá cómo los trata Java. De hecho, solo hay algunas clases y métodos que se necesitan conocer, ya que Java se centra en hilos, no en procesos (cada aplicación principal de Java es un hilo de hecho). Sin embargo, hay algunas funcionalidades adicionales que permiten llamar a programas externos o crear procesos desde una aplicación Java.

1.1. Crear procesos

Para crear un proceso en Java, es necesario obtener un objeto ***Process***. Esto se puede lograr de dos maneras diferentes:

- Utilizar la clase ***ProcessBuilder***. Es necesario crear un *array* de *strings* con el nombre del programa a ejecutar y sus argumentos, luego, llamar al método *start()*.

```
String[] cmd = {"ls", "-l"};
ProcessBuilder pb = new ProcessBuilder(cmd);
Process p = pb.start();
```

- Utilizar la clase ***Runtime***. También será necesario crear un *array* de *strings* con el nombre del programa y sus argumentos, y luego se llamará al método *exec()* con ese *array* como parámetro.

```
String[] cmd = {"notepad.exe"};
Runtime rt = Runtime.getRuntime();
Process p = rt.exec(cmd);
```

En ambos casos, se está ejecutando un comando o programa existente en el sistema operativo donde Java se está ejecutando actualmente. Puede ser un *shellscript* de Linux, un archivo *exe*, o incluso otra aplicación Java a través de un comando Java. Si no se puede encontrar el programa, o no tiene permiso para ejecutarlo, se lanzará una excepción cuando se intente llamar a los métodos *start()* o *exec()* de las clases *ProcessBuilder* o *Runtime*, respectivamente. Esta excepción será un subtipo de *IOException*.

```
try {
    Process p = pb.start();
    …
} catch (IOException e) {
    System.err.println("Exception: " + e.getMessage());
    System.exit(-1);
}
```

1.1.1. Diferencias entre ProcessBuilder y Runtime

Quizás te preguntes… ¿por qué hay dos formas de hacer lo mismo? Bien, *Runtime* pertenece al núcleo de Java desde su primera versión, mientras que *ProcessBuilder* se añadió en Java 5. Con *ProcessBuilder* se pueden añadir variables de entorno y cambiar el directorio de trabajo actual para que se inicie el proceso. Estas funciones no están disponibles para la clase *Runtime*. Además, existen algunas diferencias sutiles entre estas

dos clases. Por ejemplo, la clase *Runtime* permite ejecutar un comando pasando toda la cadena como argumento, sin dividirla en argumentos separados en un *array*:

```
Process p = Runtime.getRuntime.exec("ls -l");
```

1.2. Sincronizar procesos

Acabas de aprender a crear y ejecutar un proceso en Java. Después de llamar a los métodos *start()* o *exec()*, el programa Java continúa y ejecuta su siguiente instrucción. Si necesitas que se detenga hasta que el sub-proceso finalice su tarea, se puede llamar al método ***waitFor()*** del objeto *Process* creado. Esto hace que el programa principal se detenga hasta que se complete el proceso.

Llamar al método *waitFor()* puede lanzar una *InterruptedException* si el sub-proceso se ha interrumpido inesperadamente. Si todo está bien, el control vuelve a la aplicación principal de Java tan pronto como finalice la tarea.

```
try {
    Process p = pb.start();
    p.waitFor();
    …
} catch (IOException e) {
    System.err.println("Exception: " + e.getMessage());
    System.exit(-1);
} catch (InterruptedException e) {
    System.err.println("Interrupted: " + e.getMessage());
}
```

El método *waitFor()* devuelve un valor entero. Este valor suele ser 0 cuando el proceso ha finalizado correctamente y cualquier otro número si finalizó inesperadamente. Entonces se puede verificar el estado final de un proceso comparando su valor de retorno:

```
int value = p.waitFor();
if (value != 0)
    System.out.println("The task finished unexpectedly");
```

1.3. Finalizar procesos

Se puede finalizar un proceso creado previamente en el programa llamando al método *destroy()*. Al hacer esto, el recolector de basura Java liberará todos los recursos asociados a ese proceso.

```
ProcessBuilder pb = new ProcessBuilder(…)
Process p = pb.start();
…
p.destroy();
```

1.4. Comunicarse con los procesos

Por lo general, un proceso necesita obtener cierta información (del usuario o de un archivo, por ejemplo) y generar algunos resultados (en un archivo, en una pantalla, etc). En muchos sistemas operativos, cuando un proceso utiliza una entrada/salida determinada, sus hijos

utilizan la misma entrada/salida. En otras palabras, si un proceso está leyendo datos de un archivo como entrada estándar y crea un sub-proceso, este sub-proceso también tendrá el mismo archivo como entrada predeterminada.

Sin embargo, Java no tiene tal comportamiento. Cuando se crea un proceso en Java a partir de otro proceso (padre), tiene su propia interfaz de comunicación. Si es necesario comunicarse con este sub-proceso, habrá que obtener sus flujos de entrada y salida. Al hacer esto, se podrá enviar datos a ese sub-proceso desde su proceso principal y también obtener los resultados de su proceso principal.

El siguiente ejemplo obtiene el resultado del sub-proceso y lo imprime en la pantalla:

```
Process p = pb.start();
BufferedReader br = new BufferedReader(new InputStreamReader(p.getInputStream()));

String line = "";
System.out.println("Process output: ");

while ((line = br.readLine()) != null) {
    System.out.println(line);
}
```

Hay algo que debes saber cuando manejas los datos del proceso. Algunos sistemas operativos (como Linux, Android, Mac OS X, etc) utilizan UTF-8 como formato de codificación, mientras que otros sistemas (Windows) utilizan su propio formato de codificación. Esto puede ser un problema si, por ejemplo, se guarda un archivo de texto en Linux y se lee desde Windows. Para evitar estos problemas, se puede utilizar un segundo argumento al crear el objeto *InputStreamReader*, para decirle a la JVM cuál es el formato de codificación esperado para la entrada:

```
BufferedReader br = new BufferedReader(
new InputStreamReader(p.getInputStream(), "UTF-8"));
```

1.5. Ejemplo

Este ejemplo crea un proceso para llamar al comando “ls” (se espera que se ejecute en Linux o Mac OS X), con la opción “-l” para tener una lista detallada de archivos y carpetas del directorio. Luego, captura la salida y la imprime en la consola (o salida estándar).

```
import java.io.*;

public class FolderListing {

    public static void main(String[] args) {
        String[] cmd = {"ls", "-l"};
        String line = "";
        ProcessBuilder pb = new ProcessBuilder(cmd);

        try {
            Process p = pb.start();
            BufferedReader br = new BufferedReader(new InputStreamReader(p.getInputStream()));

            System.out.println("Process output:");
```

```
            while ((line = br.readLine()) != null) {
                System.out.println(line);
            }
        } catch (Exception e) {
            System.err.println("Exception:" + e.getMessage());
        }
    }
}
```

Ejercicio opcional 1

Crea un proyecto llamado ***ProcessListPNG*** con un programa que le pida al usuario que introduzca una ruta (por ejemplo, */myfolder/photos*), y luego inicia un proceso que imprime una lista de todas las imágenes PNG encontradas en esta ruta. Intenta hacerlo de forma recursiva (ya sea con un comando del sistema operativo o con tu propio *script*).

Ejercicio opcional 2

Crea un proyecto llamado ***ProcessKillNotepad*** con un programa que inicie el bloc de notas o cualquier editor de texto similar desde su sistema operativo. Luego, el programa esperará 10 segundos a que finalice el sub-proceso y, transcurrido ese tiempo, se destruirá. Para dormir 10 segundos, utiliza esta instrucción:

```
Thread.sleep(milliseconds);
```

2. Prioridades de hilos

Cuando se tienes varios hilos ejecutándose en un programa, se puede cambiar la prioridad de cada hilo, de modo que aquellos hilos con mayor prioridad recibirán el procesador con más frecuencia. Esta característica solo se aplica a hilos, no a procesos, ya que la JVM no es responsable de los procesos externos.

Las prioridades en los hilos son solo números enteros de 1 (almacenado en la constante ***Thread.MIN_PRIORITY***) a 10 (almacenado en la constante ***Thread.MAX_PRIORITY***). De forma predeterminada, cada hilo tiene una prioridad de 5 (constante ***Thread.NORM_PRIORITY***), y cada hilo heredará la prioridad de su padre, a menos que se cambie más tarde.

Para cambiar la prioridad de un hilo, se puede usar su método ***setPriority()***, pasando un número entero como parámetro. También se puede comprobar la prioridad del hilo con ***getPriority()***.

```
Thread t1 = new MyThread();
Thread t2 = new MyThread();
Thread t3 = new MyThread();

t1.setPriority(Thread.MIN_PRIORITY);
t2.setPriority(Thread.NORM_PRIORITY);
t3.setPriority(Thread.MAX_PRIORITY);

System.out.println("Priority of thread #2 is "  + t2.getPriority());
```

2.1. Dependencia del sistema operativo

Existe un problema con las prioridades según el sistema operativo que se esté usando. En los sistemas Windows, verás que tus hilos se comportan más o menos de acuerdo con sus prioridades, pero en sistemas Linux y Mac OS X, la prioridad que se intenta establecer para cada hilo no tiene ningún efecto. Así que se tendrá en cuenta que el comportamiento esperado de los hilos no está garantizado, y dependerá del sistema operativo, salvo que se busquen otras opciones.

2.1.1. Otras opciones para establecer prioridades: números aleatorios y yield/sleep

Si es necesario estar seguro de que algunos hilos tendrán una prioridad más alta, no se puede confiar en el método ***setPriority()***, porque el sistema operativo puede ignorar estas prioridades. Una opción alternativa es utilizar números aleatorios y los métodos *yield/sleep* para forzar a los hilos a liberar el procesador de acuerdo con su prioridad real. Echa un vistazo al siguiente ejemplo:

```
public class MyPrioritizedThread extends Thread {
    int priority;
    public MyPrioritizedThread(int priority) {
        this.priority = priority;
    }
```

```
    @Override
    public void run() {
        java.util.Random r = new java.util.Random(System.currentTimeMillis());

        while (condition) {
            // Generate a random number between 1 and 10
            int number = r.nextInt(10) + 1;

            // If this number is greater or equal than thread's
            // priority, yield
            if (number >= priority)
                Thread.yield();

            … // Rest of the instructions for our run loop
        }
    }
}
```

Se define la subclase *Thread* con su propio atributo *priority*, que será administrado desde el código. En el método *run()*, se genera un número aleatorio entre 1 (que corresponderá a *Thread.MIN_PRIORITY*) y 10 (que corresponderá a *Thread.MAX_PRIORITY*). Si este número es mayor o igual que el atributo *priority*, entonces el hilo cederá (*yield*). Fíjate que los hilos con prioridades más bajas (es decir, más cerca de 1) cederán con más frecuencia, y los hilos con prioridades más altas (es decir, más cerca de 10) cederán de vez en cuando.

Si el planificador de tareas ignora la instrucción *yield* y las prioridades parecen no tener ningún efecto, entonces cambia la instrucción *yield* por tiempo para dormir:

```
if (number >= priority)
try {
    Thread.sleep(5);
} catch (Exception e) { }
```

Ejercicio opcional 3

Crea un proyecto llamado ***ThreadRacePriorities*** basado en el proyecto *ThreadRace* del ejercicio 3. Modifica el código para que el hilo A tenga *MAX_PRIORITY*, hilo B tenga *NORM_PRIORITY* y el hilo C tenga *MIN_PRIORITY*. Hazlo con el método *setPriority()*. Intenta probar o ver los resultados en diferentes sistemas operativos.

Ejercicio opcional 4

Crea un proyecto llamado ***ThreadRacePrioritiesRandom*** que cambie la asignación de prioridades del ejercicio anterior para la segunda opción explicada (números aleatorios y métodos *yield/sleep*). Intenta probar o ver los resultados en diferentes sistemas operativos.

3. El problema productor-consumidor

El problema productor-consumidor es un problema clásico en la programación concurrente. En este tipo de problemas, se tiene un *buffer* de datos, algunos productores que colocan datos en ese *buffer* y algunos consumidores que toman datos del *buffer*. Hay que asegurarse que los consumidores no intenten coger datos cuando el *buffer* esté vacío y, en algunos casos, que los productores no produzcan más datos hasta que los consumidores hayan cogido el existente o si el *buffer* ya está lleno.

En este tipo de problemas el uso de la palabra clave *synchronized* no es suficiente. Deben añadirse algunos mecanismos para que los productores o los consumidores esperen hasta que la otra parte haya hecho su trabajo. Para hacer esto, se pueden usar los métodos ***wait()***, ***notify()*** y ***notifyAll()***, desde la clase *Objet*:

El método ***wait()*** se puede llamar desde dentro de un bloque *synchronized*. Luego, la JVM pone el hilo en suspensión y libera el objeto controlado por este bloque sincronizado, de modo que otros hilos que ejecutan bloques sincronizados del mismo objeto pueden continuar.

Los métodos ***notify()*** y ***notifyAll()*** son llamados por un hilo que ha terminado su tarea dentro de una sección crítica, antes de salir de ella, para decirle a la JVM que puede despertar un hilo previamente puesto en suspensión con el método *wait()*. La principal diferencia entre estos dos métodos (*notify()* y *notifyAll()*) es que, con *notify()*, la JVM elige un hilo en espera (aleatoriamente), mientras que con *notifyAll()* la JVM despierta a todos los hilos en espera, y el primero que entra en la sección crítica es el que continúa (los demás seguirán esperando).

Observa el siguiente ejemplo: se crearán dos tipos de hilos: un *Producer* que colocará datos (por ejemplo, un número entero) en un objeto dado (se llamará *SharedData*), y un *Consumer* que obtendrá los datos. La clase *SharedData* será la que se muestra:

```
public class SharedData {
    int data;

    public int get() {
        return data;
    }

    public void put(int newData) {
        data = newData;
    }
}
```

Las clases *Producer* y *Consumer* serán como se muestran a continuación:

```
public class Producer extends Thread {
    SharedData data;

    public Producer(SharedData data) {
        this.data = data;
    }
```

```
    @Override
    public void run() {
        for (int i = 0; i < 50; i++) {
            data.put(i);
            System.out.println("Produced number " + i);
            try {
                Thread.sleep(10);
            } catch (Exception e) { }
        }
    }
}

public class Consumer extends Thread {
    SharedData data;

    public Consumer(SharedData data) {
        this.data = data;
    }

    @Override
    public void run() {
        for (int i = 0; i < 50; i++) {
            int n = data.get();
            System.out.println("Consumed number " + n);
            try {
                Thread.sleep(10);
            } catch (Exception e) { }
        }
    }
}
```

La aplicación principal creará un objeto *SharedData* objeto y un hilo de cada tipo, e iniciará ambos hilos.

```
public static void main(String[] args) {
    SharedData sd = new SharedData();
    Producer p = new Producer(sd);
    Consumer c = new Consumer(sd);

    p.start();
    c.start();
}
```

Si copias este ejemplo y observas cómo funciona, verás algo como esto:

```
Produced number 0
Consumed number 0
Produced number 1
Consumed number 1
Produced number 2
Consumed number 2
Produced number 3
Consumed number 3
Produced number 4
Consumed number 4
Produced number 5
Consumed number 5
Produced number 6
Consumed number 6
Produced number 7
Consumed number 7
…
```

Fíjate como, a veces, el productor pone números demasiado rápido y, a veces, el consumidor también obtiene números demasiado rápido, de modo que no están coordinados (el consumidor puede leer dos veces el mismo número, el productor puede poner dos números consecutivos).

Se podría pensar que, si se añade la palabra clave *synchronized* a los métodos *get()* y *put()* de la clase *SharedData*, se resolvería el problema:

```
public class SharedData {
    int data;

    public synchronized int get() {
        return data;
    }

    public synchronized void put(int newData) {
        data = newData;
    }
}
```

Sin embargo, si ejecutas el programa nuevamente, podrás notar que aún falla:

```
Produced number 0
Consumed number 0
Consumed number 0
Produced number 1
Produced number 2
Consumed number 2
Produced number 3
Consumed number 3
Produced number 4
Consumed number 4
Produced number 5
Consumed number 5
Produced number 6
…
```

De hecho, hay dos problemas que deben resolverse. Se empezará por el más importante de ellos: el productor y el consumidor tienen que trabajar coordinados: tan pronto como el productor ponga un número, el consumidor puede obtenerlo y el productor no podrá producir más números hasta que el consumidor obtenga el número anterior.

Para hacer esto, se necesitan añadir algunos cambios a la clase *SharedData*. En primer lugar, será necesario una bandera que indique a los productores y consumidores quién es el siguiente. Dependerá de si hay nuevos datos para consumir (turno para el consumidor) o no (turno para el productor).

```
public class SharedData {
    int data;
    boolean available = false;

    public synchronized int get() {
        available = false;
        return data;
    }

    public synchronized void put(int newData) {
        data = newData;
```

```
        available = true;
    }
}
```

Además, deberá asegurarse que los métodos *get()* y *put()* se llamarán alternativamente. Para hacer esto, se necesitará usar la bandera booleana y los métodos *wait()* y *notify()*/*notifyAll()*, de la siguiente forma:

```
public class SharedData {
    int data;
    boolean available = false;

    public synchronized int get() {
        if (!available)
            try {
                wait();
            } catch (Exception e) { }
        available = false;
        notify();

        return data;
    }

    public synchronized void put(int newData) {
        if (available)
            try {
                wait();
            } catch (Exception e) {  }
        data = newData;
        available = true;
        notify();
    }
}
```

Observa como se utilizan los métodos *wait()* y *notifyAll()*. Con respecto al método *get()* (llamado por el *Consumer*), si no hay nada disponible, espera. Luego, obtiene el número, estableciendo la bandera a falso de nuevo y notifica al otro hilo. El método *put()* (llamado por el *Producer*), si hay algo disponible, espera hasta que alguien notifique el cambio. Luego, configura los nuevos datos, cambia la bandera a verdadero de nuevo y notifica al otro hilo.

Si ambos hilos intentan llegar a la sección crítica al mismo tiempo, el *Consumer* tendrá que esperar (la bandera está configurada en falso al inicio), y *Producer* establecerá los primeros datos que se consumirán. A partir de ahí, se alternan en la sección crítica, consumiendo y produciendo nuevos datos cada vez.

```
Produced number 0
Consumed number 0
Produced number 1
Consumed number 1
Produced number 2
Consumed number 2
Produced number 3
Consumed number 3
Produced number 4
Consumed number 4
Consumed number 5
Produced number 5
```

Ejercicio opcional 5

Crea un proyecto llamado ***DishWasher***. Se simulará un proceso de lavado de platos en casa, cuando alguien lava los platos y otra persona los seca. Crea las siguientes clases:

- Una clase ***Dish*** con solo un atributo entero: el número del plato (para identificar los diferentes platos).
- Una clase ***DishPile*** que almacenará hasta 5 platos. Tendrá método *wash()* que pondrá un plato en la pila (si hay espacio disponible), y un método *dry()* que cogerá un plato de la pila (si hay alguno). Tal vez necesites un parámetro *Dish* en el método *wash()*, para añadir un plato a la pila.
- Un hilo ***Washer*** que recibirá un número N como parámetro, y un objeto *DishPile*. En su método *run()*, pondrá (*wash*) N platos en la pila, con una pausa de 50 ms entre cada plato.
- Un hilo ***Dryer*** que recibirá un número N como parámetro, y un objeto *DishPile*. En su método *run()*, sacará (*dry*) N platos de la pila, con una pausa de 100 ms entre cada plato.
- La clase principal creará el objeto *DishPile*, y un hilo de cada tipo (*Washer* y *Dryer*). Tendrán que lavar/secar 20 platos de forma coordinada, para que la salida sea algo como esto:

```
Washed dish #1, total in pile: 1
Drying dish #1, total in pile: 0
Washed dish #2, total in pile: 1
Drying dish #2, total in pile: 0
Washed dish #3, total in pile: 1
Washed dish #4, total in pile: 2
Drying dish #4, total in pile: 1
Washed dish #5, total in pile: 2
Washed dish #6, total in pile: 3
Drying dish #6, total in pile: 2
Washed dish #7, total in pile: 3
Washed dish #8, total in pile: 4
Drying dish #8, total in pile: 3
Washed dish #9, total in pile: 4
Washed dish #10, total in pile: 5
Drying dish #10, total in pile: 4
```