Evaluable II

Óscar Lestón Casais
Danarys Michel Constantine

CDI_5

ÍNDICE

1. Introducción

Un ejemplo de coordinación de multiprocesos es la problemática presente en la tarea del productor-consumidor. Por un lado, el productor es el encargado de generar un producto para después almacenarlo. El consumidor, por otro lado, será el que tome esos productos, uno a uno y simultáneamente. Ambos procesos comparten un buffer de tamaño limitado, por lo cual hay que cerciorarse de que el productor no añada más productos de los que permite la capacidad y que el consumidor no intente tomar un producto si el buffer está vacío.

Como marco a esta casuística se trabajará con el correo electrónico de una empresa, en la cual los empleados del departamento de RRHH pueden realizar las siguientes acciones:

- Recibir correos: se almacenan en la bandeja de entrada.
    - Nota: esto sólo es posible si la bandeja no está llena.

- Abrir y leer correos: se sacan de la bandeja de entrada y se leen.
    - Nota: si la bandeja está vacía, esta acción no se puede cumplir. Además, al leer los correos, estos deben eliminarse.

Según el estado del buffer (bandeja de correos), ambos procesos productores y consumidores, que se están ejecutando concurrentemente, pueden estar “dormidos” o “despiertos”.

Cuando un consumidor abre un correo, es decir, toma un producto, **notifica de que se pueden volver a introducir correos en la bandeja.** En caso de que la bandeja se vacíe, uno de los consumidores **duerme,** hasta que **recibe la señal de despertar** de uno de los productores que acaba de agregar un correo a la bandeja.

2. Productor/consumidor con ***wait()/notify()/notifyAll()***

El objetivo en este caso es implementar un búfer con una capacidad máxima. Para ello, se crearán tres clases:

- Writer: hilo “**Writer**”
- Reader: hilo “**Reader**”
- Buffer: se encarga de sincronizar las siguientes funciones miembros de esta clase y utilizadas por los hilos “**Writer**” y “**Reader**” concurrentemente:
    - Recibir
    - Abrir

En cuanto a las funcionalidades requeridas:

- Programa principal (**Main()**):
    - Lanzar hilo productor: “**Writer**”
    - Lanzar hilo consumidor: “**Reader**”

    - Ambos compartirán el objeto “**Buffer**” que contiene una lista enlazada de enteros
        - Nota: cada entero representa un correo distinto.

- Un productor agrega métodos al Buffer con la función “**write()**”

    - Nota: el búfer tiene una capacidad máxima que impide que se añadan más elementos de los que ésta indica. Si esto se intentase, el productor esperaría a que un consumidor (sincronizado con la condición **notFull()**) deje espacio

- Un consumidor elimina los correos de la bandeja con la función “**read()**”

    - Nota: si el búfer está vacío, el consumidor no puede leer correos. Si esto se intentase, el consumidor esperará hasta que un productor (sincronizado con la condición **notEmpty()**) genere un correo

- Ambas tareas, producir un correo y eliminarlo, se simulan con un breve tiempo de espera aleatorio. Para ello se han utilizado las clases **java.util.concurrent.ThreadLocalRandom** y/o **java.util.Random**

- Se pueden llegar a producir **bloqueos** (cuando dos o más procesos se quedan a la espera de una situación que nunca se va a dar).
    - Nota: se adjunta una captura de pantalla abajo en el trozo de código dónde se produce

3. Problema del productor/consumidor con ***Java locks***

En esta parte, modificamos lo hecho anteriormente para así poder utilizar objetos basados en **ReentrantLock()**, además de la interfaz **Condition** de Java.

Las funcionalidades requeridas se detallan a continuación:

- Se emplearán los métodos **lock()/unlock()** del objeto **Lock** que poseen el mismo comportamiento y semántica básicos que sentencias y métodos sincronizados; pero con capacidades ampliadas. De esta manera, controlamos la concurrencia al igual que con los tipos **atomic** y **synchronized**, éste último con los métodos **notify()/wait()**.

- La interfaz **Condition** combinada con implementaciones de **Lock,** convierte los métodos de monitorización (**wait()**, **notify()** y **notifyAll()**) en objetos distintos para dar el efecto de tener múltiples conjuntos de espera por objeto**.** Lo que se busca es proporcionar un medio para que un hilo concreto pueda suspender su ejecución para “esperar”, dicho coloquialmente, a ser notificado por otro hilo de que una condición de estado ahora se cumple. Para crear estas condiciones, se emplea el método **newCondition()** y en nuestro programa:

    - **noVacio:** para controlar que en la bandeja de entrada hay correos

- **noLLeno:** para controlar que la bandeja no sobrepasa su límite

4. Problema del productor/consumidor con ***Java collection objects***

El objetivo de este apartado es reemplazar la clase "**Buffer**" que se pedía al principio del proyecto por una cola de bloqueo o "**Blockingqueue**". Así pues, cuando un hilo intente agregar un elemento a una cola completa o eliminar un elemento de una cola vacía; la cola bloqueará el hilo, es decir, éste pasará a estado "en espera". El hilo se encuentra en esta situación hasta que la cola ya no esté llena o ya no esté vacía.

En Java hay tres implementaciones de colas de bloqueo: **ArrayBlockingQueue** (la empleada en este caso), **LinkedBlockingQueue** y **PriorityBlockingQueue**.

5. Figuras (gráficas, imágenes, etc) y conclusiones

Figura 1 - Interbloqueo: Captura de pantalla de la situación donde se producen interbloqueos entre varios procesos

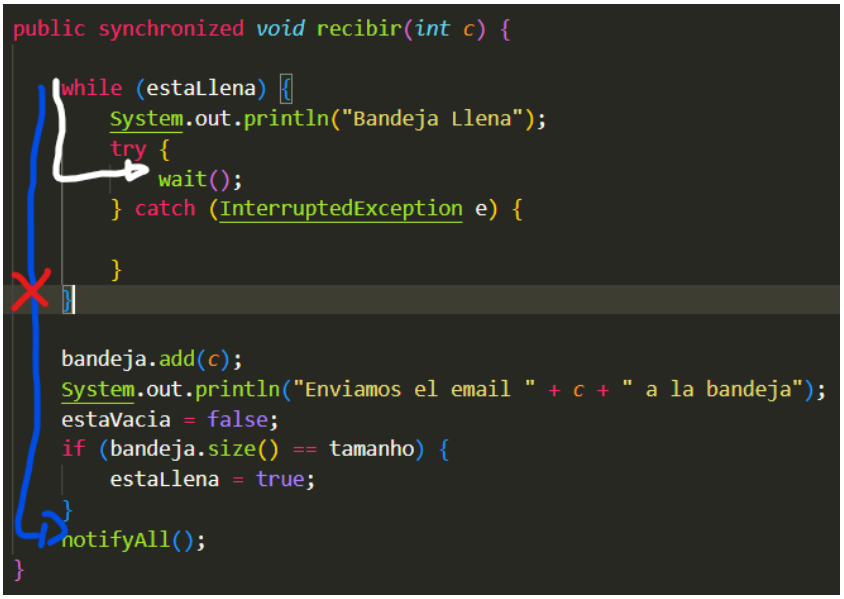

Figura 2 - Uso de JavaLocks para el problema Productor/Consumidor

- **Linux: (falta grafica)**

- **Conclusión:** *Por un lado, con el tipo "Atomic" recae en Java toda la responsabilidad a la hora de comprobar cuándo se produce indeterminismo y evitarlo. La ventaja es que se logra un código pequeño y muy legible. Por otro lado, con el tipo "Synchronized" se necesitan más líneas de código y la sección crítica está controlada por el programador, motivo por el cual siempre cabe la posibilidad de que*

se produzcan interbloqueos entre distintos synchronized. Finalmente, utilizando en este caso ReentrantLock(), se instancia un objeto llamado ***lock*** *que funciona, como el nombre indica, como un candado sobre sí mismo y cuya apertura se realiza con la llamada al método* ***unlock()****. En el sentido de lograr atomicidad en la ejecución del código funciona igual que* ***synchronized().*** *Podemos concluir que la ventaja en la usanza de ReentrantLock es para los propios desarrolladores, porque pueden decidir cuándo termina la sección crítica en la ejecución del código. Además, en la propia API de Java ya se ven detalladas las características a mayores que posee con respecto a los otros tipos. (p.e: saber qué hilo se está ejecutando en cada momento, instanciar una cola de hilos, saber cuántos hay, etc). Algo curioso es que, si no se utiliza* ***unlock()*** *el programa nunca va a terminar por su cuenta, se debe interrumpir manualmente. De igual forma, se puede decir que la interfaz* ***Condition*** *de Java sustituye a los métodos de monitorización de objetos. De la misma manera, es interesante que la información de estado (p.e: las propias condiciones) esté compartida, es decir, se produce en diferentes hilos. Por esta razón, las condiciones tienen bloqueos asociados, pero la diferencia está en que gracias a la condición, se libera atómicamente el bloqueo asociado y suspende el hilo actual (lo mismo ocurre con* ***wait()****).*

Figura 3 - Uso de Java Collection Objects para el problema del Productor/Consumidor

- **Linux: (falta grafica)**

- **Conclusión:** *BlockingQueue, del paquete java.util.concurrent, hereda tanto los métodos de la interfaz Queue como los de Collection. Cuando la operación solicitada no se puede ejecutar, cada método se comportara de manera diferente. Con la opción elegida, ArrayBlockingQueue, la cola de bloqueo se basa en el principio FIFO para ordenar los elementos. Además, el "lock" es el mismo para el productor como para el consumidor, o lo que es lo mismo, se utiliza el mismo bloqueo para recibir un correo en la bandeja como para abrirlo. Por otro lado, por el propio funcionamiento de la cola, a medida que los correos se van recibiendo o abriendo, también se van añadiendo o eliminando; lo cual es más eficiente.*