Concurrencia

El problema del productor / consumidor con buffer acotado

En este tipo de problemas, hay un buffer entre dos procesos llamados productor y consumidor. El productor genera ítems que deposita en el buffer y el consumidor captura ítems del buffer y los procesa. Por simplicidad, vamos a suponer que los ítems de tipo `double`. Como el buffer es compartido, cada proceso debe acceder al buffer en exclusión mutua. Vamos a emplear un array de tipo `double` y de tamaño `N` (para no hacerlo muy grande, asumimos de tamaño 8).

El buffer debe de tener dos punteros, `inBuf` y `outBuf`, los cuales apuntan a los índices del array para depositar y capturar ítems, respectivamente. La variable `counter` lleva cuenta del número de ítems almacenados actualmente en el buffer. La siguiente figura muestra el buffer como un array circular para el depósito y captura de un ítem con los punteros anteriormente mencionados.

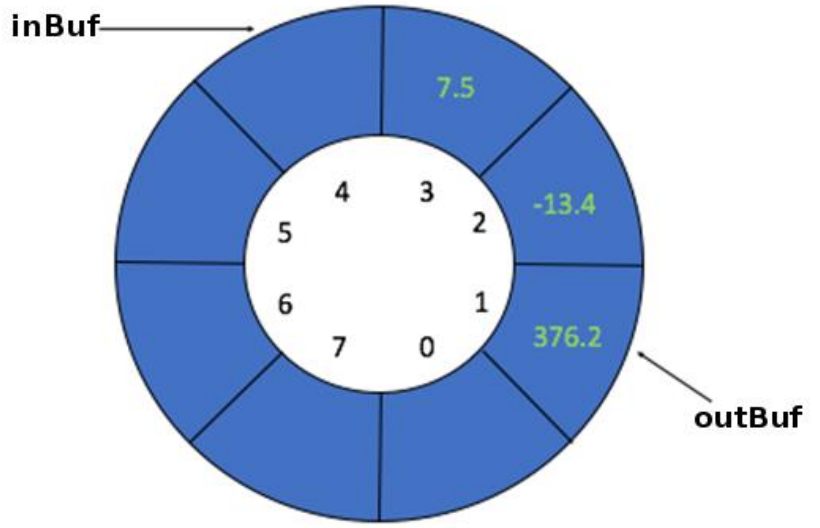

Figura 1. Un buffer compartido implementado como un *array* circular

En este problema, vemos que además de la exclusión mutua, hay dos restricciones adicionales que hay que satisfacer:

1. El consumidor no debe de tomar ítems del buffer cuando éste está vacío
2. El productor no debe de depositar nuevos ítems en el buffer si éste está lleno. El buffer se puede llenar si el productor deposita ítems en el buffer a mayor velocidad que el consumidor toma los ítems del mismo.

Esta forma de sincronización se denomina sincronización condicional. Requiere de un proceso que espere al cumplimiento de una condición (como que el buffer esté no vacío) antes de ejecutar sus operaciones. La clase `EnclosedBuffer` se muestra a continuación, emplea un semáforo de exclusión mutua para garantizar que todas las variables compartidas se acceden de esa manera. Además, el semáforo de conteo `isFull` se usa para hacer que un productor espere en caso de que el búfer esté lleno, y el semáforo `isEmpty` se usa para hacer que un consumidor espere cuando el búfer está vacío.

```
1  import java.util.concurrent.Semaphore;
2  class EnclosedBuffer {
3   final int size = 10;
4   double[] buffer = new double[size];
5   int inBuf = 0, outBuf = 0;
6   CountingSemaphore mutex = new CountingSemaphore(1);
```

```
7   CountingSemaphore isEmpty = new CountingSemaphore(0);
8   CountingSemaphore isFull = new CountingSemaphore(size);
9
10  public void put(double value) {
11    isFull.acquire(); // esperar si el buffer está lleno
12    mutex.acquire(); // asegura la exclusión mutua
13    buffer [inBuf] = value; // actualiza el buffer
14    inBuf = (inBuf + 1)% size;
15    mutex.release();
16    isEmpty.release(); // notifica al consumidor a la espera
17  }
18  public double get() {
19    double value;
20    isEmpty.acquire(); // espera si el buffer está vacío
21    mutex.acquire(); // asegura la exclusión mutua
22    value = buffer[outBuf]; //lectura del buffer
23    outBuf = (outBuf + 1) % size;
24    mutex.release();
25    isFull.release(); // notifica al productor a la espera
26    return value;
27  }
28 }
```

En el método depositar, la línea 11, verifica si el buffer está lleno. Si lo es, el proceso que realiza la llamada espera utilizando el semáforo `isFull`. Ten en cuenta que este semáforo se ha inicializado con el tamaño del buffer y, por lo tanto, en ausencia de un consumidor, las primeras llamadas `isFull.acquire()` no se bloquean. En este punto, el buffer estará lleno y cualquier llamada a

`isFull.acquire()` se bloqueará. Si la llamada a `isFull.acquire()` no se bloquea, entramos en la sección crítica para acceder al buffer compartido. La llamada `mutex.acquire()` en la línea 12 sirve como entrada a la sección crítica, y `mutex.release()` sirve como la salida de la sección crítica. Una vez dentro de la sección crítica, depositamos el valor en el buffer usando el puntero `inBuf` en la línea 13. La línea 16 realiza una llamada a `isEmpty.release()` para despertar a cualquier consumidor que pueda estar esperando a que el buffer esté vacío. El método `get()` es dual del método depósito.

Ejercicio 1

Las clases anteriores emplean la clase `Semaphore`. Implementa la clase `Producer` y `Consumer` que creen hilos que depositen y recuperen datos, respectivamente, durante un máximo de 30 iteraciones. Entre iteración e iteración, el productor duerme 200 ms. Y el consumidor 50 ms. Por otro lado, el productor generará números aleatorios mediante el empleo de la clase `Random`.

Ejercicio 2 (a entregar)

Una de las características más importantes de Java es que cada objeto puede ser un monitor mediante el uso de la palabra reservada **`synchronized`** en sus métodos. Si queremos obtener sincronización condicional, Java proporciona:

1. `wait()`: el cual inserta el hilo que lo invoca en la cola de espera.
2. `notify()`: despierta a un hilo de la cola de espera.
3. `notifyAll()`: despierta a todos los hilos en la cola de espera.

Realiza una implementación del problema anterior empleando exclusivamente los monitores que proporciona Java.