Programación de Redes y Servicios
Grado en Ingeniería de Tecnologías y Servicios de Telecomunicación
Universidad de Zaragoza, curso 2019/2020

Práctica 2: **Semáforos en Java**

Objetivos y requisitos

Objetivos

- Utilización de semáforos para la sincronización entre procesos concurrentes.
- Análisis, diseño e implementación de programas concurrentes.
- Verificación de la corrección de programas concurrentes.

Requisitos

- Compilador y máquina virtual de Java (JVM) Java Platform Standard Edition 7[1]
- `Eclipse`, como entorno de desarrollo[2].

1. Semáforos para el Problema de la Sección Crítica

Dijkstra propuso el mecanismo de los *semáforos* para resolver el problema de la sección crítica y evitar las esperas activas en el año 1965 [**?**]. Un semáforo tiene dos campos (variables):

- su valor (v); y
- una lista de procesos bloqueados (l).

Además, los semáforos tienen asociados dos operaciones `wait()` y `signal()` que modifican el valor y la lista de procesos bloqueados. En concreto, la operación de `wait()` puede o bien permitir la ejecución o bien bloquear el proceso que la invoca, en función del valor

[1] Disponible para descarga en `http://www.oracle.com/technetwork/java/javase/downloads/index.html`.
[2] Disponible para descarga en `http://www.eclipse.org/downloads/`

v del semáforo, mientras que la operación de `signal()` se encarga o bien de sacar de la lista de procesos bloqueados a alguno de ellos (si es no vacía), o bien de incrementar el valor v del semáforo. Más formalmente:

Procedimiento *s.wait()*
  si $s.v > 0$ **entonces** `/* El valor del semáforo es positivo */`
    $s.v = s.v - 1$
  si_no `/* El valor del semáforo es cero */`
    Bloquea la ejecución del proceso que invoca
  fin
fin

Procedimiento *s.signal()*
  si *hay procesos bloqueados en s.l* **entonces**
    Despierta a uno de ellos para que continúe su ejecución
  si_no `/* No hay procesos bloqueados en el semáforo */`
    $s.v = s.v + 1$
  fin
fin

Usualmente, las implementaciones de semáforos gestionan los procesos bloqueados como una cola con política *first-in-first-out* (FIFO). En Java, la clase `Semaphore` está definida en el paquete `java.util.concurrent` con algunas peculiaridades. El campo valor del semáforo v se denomina `permits`, y designa el número de accesos concurrentes. Este valor es un parámetro del constructor de la clase. Además, puede ser inicializado con un valor negativo. Las operaciones `wait()` y `signal()` corresponden a los métodos `acquire()` y `release()`, respectivamente. El método `acquire()` puede generar una excepción `InterruptedException` que debe ser capturada o elevada (recuerda la introducción a Programación Orientada a Objetos de la Práctica 1).

Existe otro constructor de la clase `Semaphore` que incluye un parámetro adicional `fair`, de tipo booleano, para indicar si es un *semáforo fair* o no. Un semáforo *fair* hace que el acceso en exclusión mutua sea en el orden de llamada, siguiendo una política FIFO. Cuando el parámetro `fair` es falso, el semáforo puede dar acceso a cualquier hilo que solicite el acceso antes que aquellos que estaban previamente esperando. En algunos escenarios, eso podría ser válido, pero en otros casos eso puede generar problemas de **inanición**. Por defecto, el comportamiento de los semáforos en Java es *non-fair*.

La documentación de ayuda en línea de Java sobre la clase `Semaphore` puedes encontrarla aquí: `https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Semaphore.html`.

2. Patrones básicos de concurrencia

En esta sección vas a aprender cómo se implementan en Java los distintos patrones básicos de concurrencia para resolver el problema del acceso a la sección crítica.

2.1. Patrón Signalling

El patrón *Signalling* es el patrón más sencillo de todos los patrones de concurrencia. Consiste en que la instrucción a_1 de un proceso $\mathcal{A}$ se ejecuta antes que otra instrucción b_2 de un proceso $\mathcal{B}$. En el Código 1 encontrarás el código del proceso $\mathcal{A}$.

Código 1: Ejemplo de código de Proceso $\mathcal{A}$ en el patrón *Signalling* (fichero *CodigoA.java*).

```
1  // Fichero CodigoA.java
2  import java.util.concurrent.*;
3
4  public class CodigoA implements Runnable {
5      private Semaphore _signalling;
6
7      public CodigoA(Semaphore signalling){
8          _signalling = signalling;
9      }
10
11     public void run() {
12         System.out.println("Esta instrucción va primero"); //a1
13         _signalling.release();  // a2
14     }
15
16     public static void main(String args[]) { // solo hay un main
17         Semaphore signalling = new Semaphore(0, true);
18         CodigoA codigoA = new CodigoA(signalling);
19         CodigoB codigoB = new CodigoB(signalling); // CodigoB por implementar
20         Thread miThreadA = new Thread(codigoA);
21         Thread miThreadB = new Thread(codigoB);
22         miThreadA.start(); miThreadB.start();
23     }
24 }
```

Ejercicio 1.

Escribe el código correspondiente a la clase `CodigoB` del proceso $\mathcal{B}$ de manera que la instrucción b_2 de $\mathcal{B}$ sea `System.out.println("Esta instrucción va después")`. Usa como base el código proporcionado para el proceso $\mathcal{A}$.

No hay que entregar nada respecto de este ejercicio.

2.2. Patrón Mutex y Multiplex

El patrón *Mutex* permite ejecutar la sección crítica en exclusión mutua. Una generalización de este patrón es el patrón *Multiplex*, que permite como máximo K procesos simultáneamente en ejecución en una sección crítica. El Código 2 muestra cómo puede implementarse el patrón *Multiplex*.

Código 2: Ejemplo de código del patrón *Multiplex* (fichero *CodigoMultiplex.java*).

```
1  // Fichero CodigoMultiplex.java
2  import java.util.concurrent.*;
3
4  public class CodigoMultiplex implements Runnable {
5      private Semaphore _multiplex;
6      public CodigoMultiplex(Semaphore multiplex){
7          _multiplex = multiplex;
8      }
9
10     public void run() {
11         try{
12             _multiplex.acquire();    // s.wait()
13         }
14         catch(InterruptedException e){ }
15         System.out.println("Puede haber condición de carrera");
16         System.out.println("================================");
17         _multiplex.release();    // s.signal()
18     }
19
20     public static void main(String args[]) {
21         Semaphore multiplex = new Semaphore(2, true);
22         for(int i = 0; i < 50; i++){
23             CodigoMultiplex codigo = new CodigoMultiplex(multiplex);
24             Thread miThread = new Thread(codigo);
25             miThread.start();
26         }
27     }
28 }
```

Ejercicio 2.

Ejecuta el programa `CodigoMultiplex` y observa, mediante ejecuciones consecutivas, si pueden darse condiciones de carrera. El patrón *Multiplex* implementado en el Código 2 permite que haya como máximo 2 procesos ejecutando (especificado según la llamada al constructor de la línea 22) la sección crítica, que está definida entre las líneas 15 y 16. Modifica el código fuente **para que la sección crítica del programa se ejecute en exclusión mutua** (esto es, de manera que únicamente un proceso esté ejecutándola).

No hay que entregar nada respecto de este ejercicio.

Ejercicio 3.

El problema del productor consumidor con búfer finito es un ejemplo clásico de problema de sincronización de procesos. En este problema, existen dos procesos, productor $\mathcal{P}$ y consumidor $\mathcal{C}$, que comparten un búfer limitado en espacio. La tarea de $\mathcal{P}$ es generar un producto, almacenarlo y comenzar nuevamente; mientras que $\mathcal{C}$ toma (simultáneamente, durante la producción de $\mathcal{P}$) productos uno a uno. El problema consiste en que el productor no añada más productos que la capacidad del búfer y que el consumidor no intente tomar un producto si el búfer se encuentra vacío.

La idea para la solución es la siguiente: ambos procesos $\mathcal{P}$ y $\mathcal{Q}$ se ejecutan simultáneamente y se "despiertan" o "duermen" según sea el estado del búfer. Concretamente, el productor $\mathcal{P}$ agrega productos mientras quede espacio en búfer y en el momento en que se llene, se pone a "dormir". Cuando el consumidor $\mathcal{Q}$ toma un producto del búfer, notifica a $\mathcal{P}$ que puede comenzar a trabajar nuevamente (dado que se ha generado un hueco libre en el búffer). El comportamiento del consumidor $\mathcal{Q}$ es similar: en caso de que el búfer se vacíe, se pone a dormir. En el momento en que $\mathcal{P}$ agrega un producto al búfer, éste genera una señal para despertarlo.

En este ejercicio se te pide que combines los patrones anteriores (necesitarás 2 patrones *Signalling* y 1 patrón *mutex*) para resolver el problema del productor consumidor con búfer limitado. Considera que los elementos que se generan/consumen son números reales. Para este ejercicio tienes que crear 3 clases: clase `MemoriaCompartida` (que incluirá los 3 semáforos que necesitas y el búfer) y clase `Productor` y clase `consumidor`. Para generar la clase `MemoriaCompartida` recuerda el código desarrollado en la Práctica 1 (sección *Variables compartidas*). En cuanto a la implementación del búfer, se recomienda usar una lista encadenada de números reales (`LinkedList<Double> _buffer`, consulta más información en la documentación en línea de Java[a]), dado que facilita la gestión de adición/eliminado y consulta de número de elementos. La clase `Productor` sigue el esquema dado en el Código 3, mientras que la clase `Consumidor` sigue el esquema del Código 4.

Puntuación: 2 puntos

Este ejercicio puede entregarse. Completa los códigos fuente de las clases `Consumidor`, `Productor` y `MemoriaCompartida` y guárdalos en un mismo directorio (carpeta) de nombre `p02e03`. *Recuerda: en este directorio guarda los códigos fuente (ficheros con extensión* `.java`*), no los ficheros objeto (extensión* `.class`*).*

[a] Disponible en la página web `https://docs.oracle.com/javase/7/docs/api/java/util/LinkedList.html`.

Código 3: Esquema de código para un productor en el problema de sincronización de procesos (fichero *Productor.java*).

```
1  // Fichero Productor.Java
2  ... // faltan paquetes
3  public class Productor implements Runnable {
4      private MemoriaCompartida _m;
5
6      public Productor(MemoriaCompartida m){
7          _m = m;
8      }
9
10     public void run() {
11         double elemento = 10000000;
12         while(elemento > 1.0){
13             ... // completar
14             elemento = Math.sqrt(elemento); // producir
15             _m._buffer.add(elemento);
16         }
17     }
18
19     public static void main(String args[]) {
20         ... // completar
21     }
22 }
```

Código 4: Esquema de código para un consumidor en el problema de sincronización de procesos (fichero *Consumidor.java*).

```
1  // Fichero Consumidor.Java
2  ... // faltan paquetes
3  public class Consumidor implements Runnable {
4      private MemoriaCompartida _m; // memoria compartida
5
6      public Consumidor(MemoriaCompartida m){
7          _m = m;
8
9      }
10
11     public void run() {
12         while(;;){ // bucle infinito
13             double elemento = _buffer.remove();
14             ... // completar
15             System.out.println("elemento = " + elemento);
16             Thread.sleep(400);
17         }
18     }
19 }
```

2.3. Patrón Turnstile

El patrón *Turnstile* (en español, *tornos*) puede utilizarse para controlar el flujo de entrada de un grupo de procesos a la sección crítica. Este patrón toma el nombre por su propio funcionamiento: permite el paso de un hilo en cada momento, mientras que bloquea en la barrera a todos los demás. Observa el Código 5 y el Código 6, correspondientes a las implementaciones de dos tipos de procesos usando este patrón.

Código 5: Esquema de código para un tipo de proceso con el patrón *Turnstile* (fichero *CodigoTurnstileA.java*).

```
1  // Fichero CodigoTurnstileA.Java
2  ... // faltan paquetes
3  public class CodigoTurnstileA implements Runnable {
4      private Semaphore _turnstile;
5
6      public CodigoA(Semaphore turnstile){
7          _turnstile = turnstile;
8      }
9
10     public void run() {
11         _turnstile.acquire();
12         _turnstile.release();
13         System.out.println("Proceso ejecutando esto");
14     }
15
16     public static void main(String args[]) {
17         Semaphore turnstile = new Semaphore(1, true);
18         CodigoB codigob = new CodigoB(turnstile);
19         codigob.start();
20
21         for(int i = 0; i < 50; i++){
22             CodigoA codigo = new CodigoA(turnstile);
23             Thread miThread = new Thread(codigo);
24             miThread.start();
25         }
26     }
27 }
```

Ejercicio 4.

Estudia el código anterior y analiza el comportamiento del patrón *Turnstile* mediante la ejecución y la modificación del Código 6. ¿Qué sucede si se comentan las líneas 11 y 12 (adquisición y liberación del semáforo)? ¿Por qué?

No hay que entregar nada respecto de este ejercicio.

Código 6: Esquema de código para un tipo de proceso con el patrón *Turnstile* (fichero *CodigoTurnstileB.java*).

```
1  // Fichero CodigoTurnstileB.Java
2  ... // faltan paquetes
3  public class CodigoTurnstileB implements Runnable {
4      private Semaphore _turnstile;
5
6      public CodigoTurnstileB(Semaphore turnstile){
7          _turnstile = turnstile;
8      }
9
10     public void run() {
11         Thread.sleep(100);
12         _turnstile.acquire();
13         Thread.sleep(10000);
14         _turnstile.release();
15     }
16 }
```

3. Ejercicios de análisis, diseño e implementación de programas concurrentes

En esta sección, se describen una serie de problemas de programas concurrentes que tienes que analizar, diseñar e implementar con los mecanismos descritos en las prácticas de esta asignatura.

Ejercicio 5.

Un taller de costura, que se dedica a confeccionar jerséis, ocupa a tres personas. Una persona está continuamente fabricando mangas, que deposita en un cesto de capacidad limitada. Cuando el cesto está lleno de mangas, la costurera deja de coser hasta que haya hueco libre. Otra persona está continuamente fabricando los cuerpos de los jerséis, que también deposita en su correspondiente cesta de capacidad limitada.
Finalmente, existe una tercera costurera, encargada de ensamblar mangas y cuerpos de jerséis, cogiendo para cada jersey, dos mangas de la cesta de mangas y un cuerpo de la cesta de cuerpos.
En este ejercicio tienes que implementar un programa concurrente en Java mediante semáforos que sincronice a estas tres personas, de forma que las dos primeras personas no avancen si su cesta está llena, y que la tercera persona no avance si le faltan piezas para hacer un nuevo jersey.

Puntuación: 4 puntos

Este ejercicio puede entregarse. Analiza este problema y diseña una solución utilizando semáforos de manera que se cumplan las condiciones de sincronización anteriores. Implementa los códigos fuente de las clases `CostureraManga`, `CostureraCuerpo`, `CostureraEnsambladora` *y* `MemoriaCompartida` *(esta última contendrá todas las variables compartidas) y guárdalos en un mismo directorio (carpeta) de nombre* `p02e05`. *Recuerda: en este directorio guarda los códigos fuente (ficheros con extensión* `.java`*), no los ficheros objeto (extensión* `.class`*).*

Ejercicio 6.

Se está diseñando un nuevo Sistema Operativo para un procesador emergente. Uno de los problemas más importantes que se ha detectado es el siguiente: el acceso para utilizar un determinado recurso hardware está custodiado por el "ensamblaje" de 2 tipos de procesos, *proceso tipo A* ($\mathcal{P}_A$) y *proceso tipo B* ($\mathcal{P}_B$). En primer lugar, hay una fase de creación de procesos: el Sistema Operativo va creando procesos $\mathcal{P}_A$ y procesos $\mathcal{P}_B$ con unas frecuencias desconocidas. Cuando se crea un proceso, se comprueba si se dan las condiciones necesarias y si éstas se dan, entonces se produce el ensamblaje. Si no se dan, el ensamblaje no se realiza y aparece entonces una fase de espera previa al ensamblaje.
Las condiciones son las siguientes:

- Si se crea un *proceso tipo A* ($\mathcal{P}_A$), entonces se comprueba que estén esperando previamente al menos otros 2 *proceso tipo B*. Si no hay esperando 2 procesos *proceso tipo B*, entonces $\mathcal{P}_A$ espera.
- Si se crea *proceso tipo B* ($\mathcal{P}_B$), entonces se comprueba que estén esperando previamente un proceso *proceso tipo A* y *proceso tipo B*. Si no se cumple la condición, entonces $\mathcal{P}_B$ espera.

El ensamblaje consiste en que los procesos simplemente invocan al método estático `assemble()` de la clase Recurso (recuerda que al ser estático, se invocará usando el nombre de la clase es decir, `Recurso.assemble()`). No es necesario que los procesos que se han ensamblado tengan que saber con quién se han ensamblado de forma explícita.
En este ejercicio tienes que diseñar e implementar un programa concurrente en Java que codifique los procesos `ProcesoA` y `ProcesoB` (como threads) y la clase `MemoriaCompartida`, donde se almacenan todas las variables compartidas.
NOTA: Para resolver este problema, intenta primero implementar el patrón barrera (según el esquema que hemos estudiado en clase de teoría) y adaptarlo para que haya dos colas de espera (2 barreras).

Puntuación: 4 puntos

Este ejercicio puede entregarse. Analiza este problema y diseña una solución utilizando semáforos de manera que se cumplan las condiciones de sincronización anteriores. Implementa los códigos fuente de las clases `ProcesoA`, `ProcesoB`, `CostureraEnsambladora` y `MemoriaCompartida` (esta última contendrá todas las variables compartidas) y guárdalos en un mismo directorio (carpeta) de nombre `p02e06`. *Recuerda: en este directorio guarda los códigos fuente (ficheros con extensión* `.java`*), no los ficheros objeto (extensión* `.class`*).*

4. Evaluación

La realización de las prácticas es por parejas. Las normas de la evaluación y entrega son las siguientes:

- Todos los programas entregados deben compilar correctamente, se valorará de forma muy negativa que no compile algún programa.
- Todos los programas entregados deben funcionar correctamente como se especifica en el problema.
- Todos los programas tienen que seguir el manual de estilo de Java propuesto por Oracle, disponible en el anillo digital docente (un 20 % de la nota estará en función de este requisito). Además de lo especificado en el manual de estilo, cada fichero fuente deberá comenzar con la siguiente cabecera:

```
/*
 * AUTOR: nombre y apellidos del autor de la práctica
 * NIA: número de identificación del alumno
 * FICHERO: nombre del fichero
 * TIEMPO: tiempo (en horas) empleado en la implementación
 * DESCRIPCIÓN: breve descripción del contenido del fichero
 */
```

Resumen de la puntuación de los ejercicios solicitados en esta práctica

Ejercicio	Puntuación
Ejercicio 3	2 puntos.
Ejercicio 5	4 puntos.
Ejercicio 6	4 puntos.

5. Instrucciones de entrega y Defensa

- Solo uno de los miembros de la pareja de prácticas realizará la entrega.
- Se entregará un único fichero comprimido `.zip`, que contendrá los códigos fuentes de todos los ejercicios solicitados (ficheros con extensión `.java`, **NO los ficheros con extensión .class**).
- Los códigos fuentes estarán en sus respectivas carpetas, indicadas en el enunciado de cada carpeta.
- La entrega se realizará a través Moodle (`http://moodle2.unizar.es`), se utilizará para ello una tarea correspondiente a la entrega de la práctica habilitada a tal efecto.
- **Fecha de entrega:**
  - Grupo `prs2` (921): 22 de marzo.

- Grupos `prs4` y `prs5` (921): 23 de marzo.
- Grupo `prs1` (921): 24 de marzo.
- Grupos `prs1`, `prs2` y `prs3` (931): 30 de marzo.

Las fechas de entrega coinciden con el día anterior al comienzo de la práctica 3 (sesión de prácticas número 5). **No se permite la entrega tardía.**