Dpto. Lenguajes y Ciencias de la Computación

# Programación avanzada II
# Laboratorio 7-Monitores - Locks

Los ejercicios de la relación pueden resolverse indistintamente utilizando métodos sincronizados o Locks. Los esqueletos para realizarlos están en el campus virtual.

1- Implementar el problema del productor-consumidor utilizando varios productores y varios consumidores con la particularidad de que el buffer en esta ocasión es NO destructivo. Esto es: cada vez que un productor coloca un elemento en el buffer, el elemento tiene que ser consumido por todos los consumidores (una sola vez por cada consumidor) antes de poder liberar la posición asociada al buffer. El resto del funcionamiento es similar al productor-consumidor normal.

Por ejemplo, si en el buffer tenemos los elementos [1,2,3,4] y 3 consumidores, se podría tener que cada consumidor ha consumido los siguientes elementos:

C1: 1, 2

C2: 1

C3: 1, 2, 3

Hasta que un elemento no ha sido consumido por todos los consumidores, la posición no se libera. En el ejemplo anterior, el elemento “1” ha sido consumido por todos los consumidores y su posición en el buffer podría ser utilizada para colocar otro elemento. El elemento “2”, por el contrario, sólo ha sido consumidor por C1 y C3 por lo que su posición todavía no puede ser liberada en el buffer.

```
Thread-4: Productor almacena 1: buffer=[1,0,0]
Thread-2: Consumidor 2 lee 1: buffer=[1,0,0]
Thread-1: Consumidor 1 lee 1: buffer=[1,0,0]
Thread-0: Consumidor 0 lee 1: buffer=[1,0,0]
Thread-3: Consumidor 3 lee 1: buffer=[0,0,0]
Thread-4: Productor almacena 2: buffer=[0,2,0]
Thread-0: Consumidor 0 lee 2: buffer=[0,2,0]
Thread-4: Productor almacena 3: buffer=[0,2,3]
Thread-3: Consumidor 3 lee 2: buffer=[0,2,3]
Thread-4: Productor almacena 4: buffer=[4,2,3]
Thread-1: Consumidor 1 lee 2: buffer=[4,2,3]
Thread-2: Consumidor 2 lee 2: buffer=[4,0,3]
Thread-4: Productor almacena 5: buffer=[4,5,3]
Thread-3: Consumidor 3 lee 3: buffer=[4,5,3]
Thread-0: Consumidor 0 lee 3: buffer=[4,5,3]
Thread-2: Consumidor 2 lee 3: buffer=[4,5,3]
Thread-0: Consumidor 0 lee 4: buffer=[4,5,3]
Thread-1: Consumidor 1 lee 3: buffer=[4,5,0]
Thread-4: Productor almacena 6: buffer=[4,5,6]
Thread-1: Consumidor 1 lee 4: buffer=[4,5,6]
Thread-2: Consumidor 2 lee 4: buffer=[4,5,6]
Thread-3: Consumidor 3 lee 4: buffer=[0,5,6]
Thread-4: Productor almacena 7: buffer=[7,5,6]
Thread-0: Consumidor 0 lee 5: buffer=[7,5,6]
Thread-3: Consumidor 3 lee 5: buffer=[7,5,6]
Thread-2: Consumidor 2 lee 5: buffer=[7,5,6]
Thread-1: Consumidor 1 lee 5: buffer=[7,0,6]
Thread-4: Productor almacena 8: buffer=[7,8,6]
Thread-3: Consumidor 3 lee 6: buffer=[7,8,6]
Thread-0: Consumidor 0 lee 6: buffer=[7,8,6]
Thread-2: Consumidor 2 lee 6: buffer=[7,8,6]
Thread-1: Consumidor 1 lee 6: buffer=[7,8,0]
Thread-4: Productor almacena 9: buffer=[7,8,9]
Thread-0: Consumidor 0 lee 7: buffer=[7,8,9]
Thread-1: Consumidor 1 lee 7: buffer=[7,8,9]
Thread-3: Consumidor 3 lee 7: buffer=[7,8,9]
Thread-0: Consumidor 0 lee 8: buffer=[7,8,9]
Thread-1: Consumidor 1 lee 8: buffer=[7,8,9]
Thread-2: Consumidor 2 lee 7: buffer=[0,8,9]
Thread-4: Productor almacena 10: buffer=[10,8,9]
Thread-1: Consumidor 1 lee 9: buffer=[10,8,9]
Thread-0: Consumidor 0 lee 9: buffer=[10,8,9]
Thread-0: Consumidor 0 lee 10: buffer=[10,8,9]
Thread-3: Consumidor 3 lee 8: buffer=[10,8,9]
Thread-3: Consumidor 3 lee 9: buffer=[10,8,9]
Thread-2: Consumidor 2 lee 8: buffer=[10,0,9]
Thread-1: Consumidor 1 lee 10: buffer=[10,0,9]
Thread-3: Consumidor 3 lee 10: buffer=[10,0,9]
Thread-2: Consumidor 2 lee 9: buffer=[10,0,0]
```

Thread-2: Consumidor 2 lee 10: buffer=[0,0,0]

2- Varios procesos (N) compiten por utilizar **rec** recursos en exclusión mutua. El uso correcto de los recursos es gestionado por una clase Control. Cada proceso pide un numero mayor o igual que 1 de recursos llamando al método del monitor qrecursos(id,num), donde id es el identificador del proceso que hace la petición y num es un número entero que especifica el número de recursos que pide el proceso. Después de usar los recursos, el proceso los libera ejecutando el método librecursos(id,num). Para asignar los recursos Control utiliza la técnica FCFS (First Come, First Serve) que significa que los procesos son servidos siempre en el orden en el que han realizado sus peticiones. Así un proceso que pide num recursos debe esperarse si hay otros procesos esperando aunque haya num recursos disponibles en el sistema. Implementa la clase Control, y varias hebras usuarios que realicen las peticiones a Control. La siguiente salida por pantalla muestra una posible ejecución del sistema con 5 recursos y 5 procesos.

Thread-3: Proceso 3 pide 3 recursos. Esperando 1
Thread-3: Proceso 3 coge 3 recursos. Quedan 2
Thread-0: Proceso 0 pide 5 recursos. Esperando 1
Thread-4: Proceso 4 pide 4 recursos. Esperando 2
Thread-2: Proceso 2 pide 3 recursos. Esperando 3
Thread-1: Proceso 1 pide 4 recursos. Esperando 4
Thread-3: Proceso 3 devuelve 3 recursos. Quedan 5
Thread-0: Proceso 0 coge 5 recursos. Quedan 0
Thread-0: Proceso 0 devuelve 5 recursos. Quedan 5
Thread-4: Proceso 4 coge 4 recursos. Quedan 1
Thread-4: Proceso 4 devuelve 4 recursos. Quedan 5
Thread-2: Proceso 2 coge 3 recursos. Quedan 2
Thread-2: Proceso 2 devuelve 3 recursos. Quedan 5
Thread-1: Proceso 1 coge 4 recursos. Quedan 1
Thread-1: Proceso 1 devuelve 4 recursos. Quedan 5

3- Se dispone de una sala donde distintas parejas tienen citas, para ello un hombre y una mujer tienen que encontrarse (sincronizarse) en la sala. Toda la sincronización del sistema se realiza en un objeto Parejas. Cuando el hombre id quiere tener una cita llama al método *llegaHombre(id:Int)*, y lo mismo para la mujer id que llama al método *llegaMujer(id:Int)* cuando quiere tener una cita.

Las condiciones de sincronización del sistema son:
- Un hombre no puede entrar en la sala si hay otro dentro
- Un hombre que está en la sala tiene que esperar a que se forme la pareja para salir
- Las condiciones de sincronización para las mujeres son las mismas.

La siguiente salida por pantalla muestra una posible ejecución del sistema con 10 mujeres y 10 hombres.

Thread-1: Mujer 1 quiere formar pareja
Thread-10: Hombre 0 quiere formar pareja
Thread-10: Se ha formado una pareja!!!
Thread-11: Hombre 1 quiere formar pareja
Thread-6: Mujer 6 quiere formar pareja
Thread-6: Se ha formado una pareja!!!
Thread-12: Hombre 2 quiere formar pareja
Thread-3: Mujer 3 quiere formar pareja
Thread-3: Se ha formado una pareja!!!
Thread-13: Hombre 3 quiere formar pareja
Thread-4: Mujer 4 quiere formar pareja
Thread-4: Se ha formado una pareja!!!
Thread-14: Hombre 4 quiere formar pareja
Thread-5: Mujer 5 quiere formar pareja
Thread-5: Se ha formado una pareja!!!
Thread-15: Hombre 5 quiere formar pareja
Thread-7: Mujer 7 quiere formar pareja
Thread-7: Se ha formado una pareja!!!
Thread-16: Hombre 6 quiere formar pareja
Thread-8: Mujer 8 quiere formar pareja
Thread-8: Se ha formado una pareja!!!
Thread-17: Hombre 7 quiere formar pareja
Thread-2: Mujer 2 quiere formar pareja
Thread-2: Se ha formado una pareja!!!
Thread-18: Hombre 8 quiere formar pareja
Thread-0: Mujer 0 quiere formar pareja
Thread-0: Se ha formado una pareja!!!
Thread-19: Hombre 9 quiere formar pareja

Thread-9: Mujer 9 quiere formar pareja

Thread-9: Se ha formado una pareja!!!

4- **El problema de la montaña rusa**. Supón que hay **n** procesos **pasajeros**, y **un** proceso **coche**. Los pasajeros esperan repetidamente para darse una vuelta en el coche, que tiene una capacidad **C<n** de pasajeros. Sin embargo, **el coche sólo da una vuelta cuando está lleno**. El coche tarda T segundos en dar una vuelta, una vez que está lleno. Después de dar una vuelta, cada pasajero da un paseo por el parque de atracciones durante un tiempo aleatorio, antes de volver a la montaña rusa para darse otra vuelta. Diseña un programa que resuelva este problema utilizando sólo **semáforos binarios** para sincronizar las hebras.
Por ejemplo, una ejecución de este sistema con un coche de capacidad C = 5 podría ser

```
Thread-3: El pasajero 2 se sube al coche. Hay 1 pasajeros.
Thread-6: El pasajero 5 se sube al coche. Hay 2 pasajeros.
Thread-5: El pasajero 4 se sube al coche. Hay 3 pasajeros.
Thread-11: El pasajero 10 se sube al coche. Hay 4 pasajeros.
Thread-12: El pasajero 11 se sube al coche. Hay 5 pasajeros.
Thread-0:       Coche lleno!!! empieza el viaje....
Thread-0:       Fin del viaje... :-(
Thread-3: El pasajero 2 se baja del coche. Hay 4 pasajeros.
Thread-6: El pasajero 5 se baja del coche. Hay 3 pasajeros.
Thread-5: El pasajero 4 se baja del coche. Hay 2 pasajeros.
Thread-11: El pasajero 10 se baja del coche. Hay 1 pasajeros.
Thread-12: El pasajero 11 se baja del coche. Hay 0 pasajeros.
```

```
Thread-2: El pasajero 1 se sube al coche. Hay 1 pasajeros.
Thread-1: El pasajero 0 se sube al coche. Hay 2 pasajeros.
Thread-4: El pasajero 3 se sube al coche. Hay 3 pasajeros.
Thread-9: El pasajero 8 se sube al coche. Hay 4 pasajeros.
Thread-7: El pasajero 6 se sube al coche. Hay 5 pasajeros.
Thread-0:       Coche lleno!!! empieza el viaje....
Thread-0:       Fin del viaje... :-(

....
```

5- Supón que hay un río cerca de la Escuela de Informática que la separa de un centro de ocio para los estudiantes. Hay estudiantes de dos tipos, los que utilizan móviles Android, y los que utilizan IPhones. Para cruzar el río, existe una barca que tiene 4 asientos, y que no se mueve hasta que no está completa (hay exactamente 4 estudiantes subidos en ella). Para garantizar la seguridad de los pasajeros, no se permite que haya un estudiante Android con tres estudiantes IPhones, ni un estudiante IPhone con tres Android. Cualquier otra configuración de estudiantes en la barca es segura. Implementa una solución a este problema utilizando métodos sincronizados o locks para gestionar la sincronización, teniendo en cuenta que deben satisfacerse las dos siguientes condiciones de sincronización:

- CS1: Un estudiante no puede subirse en la barca hasta que no hay algún asiento libre en la barca y la configuración para él es segura.

- CS2: Un estudiante no puede bajarse de la barca hasta que no se ha terminado el viaje.

Los estudiantes Android/IPhone llaman al método def android(id:Int)/def iphone(id:Int) del objeto barca cuando quieren cruzar el río. Para simplificar el problema, se supone que el último estudiante que sube a la barca (el que ocupa el cuarto asiento) es el que maneja la barca y la lleva al otro extremo del río, es decir, est estudiante es el que decide cuando se ha terminado el viaje, y avisa al resto de los ocupantes de la barca para que se bajen.

La siguiente salida por pantalla muestra una posible ejecución del sistema con 4 estudiantes Android y 4 IPhones.

```
Thread-2: Estudiante IPhone 2 se sube a la barca. Hay: iphone=1,android=0
Thread-5: Estudiante Android 1 se sube a la barca. Hay: iphone=1,android=1
Thread-3: Estudiante IPhone 3 se sube a la barca. Hay: iphone=2,android=1
Thread-4: Estudiante Android 0 se sube a la barca. Hay: iphone=2,android=2
Thread-4: Empieza el viaje....
Thread-4: fin del viaje....
```

```
Thread-4: Estudiante Android 0 se baja de la barca. Hay: iphone=2,android=1
Thread-2: Estudiante IPhone 2 se baja de la barca. Hay: iphone=1,android=1
Thread-5: Estudiante Android 1 se baja de la barca. Hay: iphone=1,android=0
Thread-3: Estudiante IPhone 3 se baja de la barca. Hay: iphone=0,android=0
Thread-0: Estudiante IPhone 0 se sube a la barca. Hay: iphone=1,android=0
Thread-1: Estudiante IPhone 1 se sube a la barca. Hay: iphone=2,android=0
Thread-7: Estudiante Android 3 se sube a la barca. Hay: iphone=2,android=1
Thread-6: Estudiante Android 2 se sube a la barca. Hay: iphone=2,android=2
Thread-6: Empieza el viaje....
Thread-6: fin del viaje....
Thread-6: Estudiante Android 2 se baja de la barca. Hay: iphone=2,android=1
Thread-0: Estudiante IPhone 0 se baja de la barca. Hay: iphone=1,android=1
Thread-1: Estudiante IPhone 1 se baja de la barca. Hay: iphone=0,android=1
Thread-7: Estudiante Android 3 se baja de la barca. Hay: iphone=0,android=0
```

6- Supongamos que N niños asisten a una fiesta de cumpleaños, en la que se les ofrece de forma repetida raciones de tarta de chocolate que se encuentran en una bandeja. Como las raciones se acaban con frecuencia, existe un pastelero que, cuando lo avisan, pone una nueva tarta en la bandeja (suponemos que cada tarta da lugar a un número fijo R de raciones). Cuando un niño quiere una ración de tarta, va a la bandeja y la coge. Si ve que la bandeja se ha quedado vacía, avisa al pastelero para que traiga una nueva tarta. Implementa este sistema utilizando métodos sincronizados, teniendo en cuenta que deben satisfacerse las dos condiciones de sincronización siguientes:
-CS1: Un niño que quiere una ración de tarta debe esperar si la bandeja está vacía.

-CS2: El pastelero no pone una nueva tarta en la bandeja, hasta que la bandeja está vacía.

Supón que el pastelero es perezoso, por lo que mientras que no lo avisan se queda durmiendo un ratito. El esqueleto del sistema se encuentra en el campus virtual. Cuando un niño quiere una ración llama al método def quieroRacion(id:Int) del objeto bandeja. Por otro lado, el pastelero llama al método def tarta() para poner una nueva tarta sobre la bandeja. Supón que la bandeja está inicialmente vacía.

La siguiente salida por pantalla muestra una posible ejecución del sistema con tartas de 5 raciones.

```
Thread-10: El pastelero pone una nueva tarta.
Thread-1: Niño 1 ha cogido una ración. Quedan 4
Thread-0: Niño 0 ha cogido una ración. Quedan 3
Thread-3: Niño 3 ha cogido una ración. Quedan 2
Thread-3: Niño 3 ha cogido una ración. Quedan 1
Thread-6: Niño 6 ha cogido una ración. Quedan 0
Thread-10: El pastelero pone una nueva tarta.
Thread-5: Niño 5 ha cogido una ración. Quedan 4
Thread-8: Niño 8 ha cogido una ración. Quedan 3
Thread-0: Niño 0 ha cogido una ración. Quedan 2
Thread-2: Niño 2 ha cogido una ración. Quedan 1
Thread-4: Niño 4 ha cogido una ración. Quedan 0
Thread-10: El pastelero pone una nueva tarta….
```

7- Supón que, por seguridad, una guardería obliga a que siempre haya al menos un adulto por cada 3 bebés, es decir, que si nBebe y nAdulto son el número de bebés y adultos en la guardería, respectivamente, siempre debe cumplirse que nBebe <= 3*nAdulto. Por lo tanto, para implementar este sistema tenemos dos condiciones de sincronización:

CS1- Un bebé que quiere entrar en la guardería no puede hacerlo hasta que sea cierta la condición *nBebe <= 3*nAdulto* con él dentro.

CS2- Un adulto que quiere salir de la guardería no puede hacerlo hasta que sea cierta la condición *nBebe <= 3*nAdulto* con él fuera de la guardería.

La siguiente salida por pantalla muestra el comienzo de una posible ejecución del sistema.

Thread-17: Ha llegado un adulto. Bebés=0, Adultos=1
Thread-14: Ha llegado un bebé. Bebés=1, Adultos=1
Thread-5: Ha llegado un bebé. Bebés=2, Adultos=1
Thread-10: Ha llegado un bebé. Bebés=3, Adultos=1
Thread-18: Ha llegado un adulto. Bebés=3, Adultos=2
Thread-2: Ha llegado un bebé. Bebés=4, Adultos=2
Thread-3: Ha llegado un bebé. Bebés=5, Adultos=2
Thread-13: Ha llegado un bebé. Bebés=6, Adultos=2
Thread-15: Ha llegado un adulto. Bebés=6, Adultos=3
Thread-6: Ha llegado un bebé. Bebés=7, Adultos=3
Thread-1: Ha llegado un bebé. Bebés=8, Adultos=3
Thread-4: Ha llegado un bebé. Bebés=9, Adultos=3 ….

8- Supón que átomos de **hidrógeno** y **oxígeno** están dando vueltas en el espacio, intentando agruparse para formar moléculas de agua. Para ello es necesario que dos átomos de hidrógeno y uno de oxígeno se sincronicen. Supongamos que cada átomo de hidrógeno y oxígeno está simulado por un proceso. La gestión de la sincronización de los átomos tiene lugar en un objeto gestor de la clase **GestorAgua**. Cada átomo de hidrógeno llama al método hListo cuando quiere formar parte de una molécula. Del mismo modo los átomos de oxígeno llaman a oListo cuando quieren combinarse con otros dos hidrógenos para formar agua. Los procesos deben esperar en estos métodos hasta que sea posible formar la molécula. Implementa una solución utilizando semáforos **binarios** que resuelva este problema. Por ejemplo, una ejecución de este programa con 10 átomos de hidrógeno y 5 de oxígeno podría ser:

Thread-4: Hidrógeno 4 quiere formar una molécula
Thread-0: Hidrógeno 0 quiere formar una molécula
Thread-12: Oxígeno 2 quiere formar una molécula
Thread-12: Molécula formada!!!
Thread-6: Hidrógeno 6 quiere formar una molécula
Thread-9: Hidrógeno 9 quiere formar una molécula
Thread-14: Oxígeno 4 quiere formar una molécula
Thread-14: Molécula formada!!!
Thread-5: Hidrógeno 5 quiere formar una molécula
Thread-13: Oxígeno 3 quiere formar una molécula
Thread-1: Hidrógeno 1 quiere formar una molécula
Thread-1: Molécula formada!!!
Thread-11: Oxígeno 1 quiere formar una molécula
Thread-3: Hidrógeno 3 quiere formar una molécula
Thread-2: Hidrógeno 2 quiere formar una molécula
Thread-2: Molécula formada!!!
Thread-10: Oxígeno 0 quiere formar una molécula
Thread-8: Hidrógeno 8 quiere formar una molécula
Thread-7: Hidrógeno 7 quiere formar una molécula
Thread-7: Molécula formada!!!
main: Fin del programa