Cuadernillo de ejercicios cortos
Concurrencia. Curso 2016–2017

Ángel Herranz Julio Mariño

Versión 1638.
Última actualización: 2017-02-07 12:39:10Z.

Este documento contiene los enunciados de los ejercicios que los alumnos deben entregar cada semana en la asignatura de Concurrencia. La entrega se realizará a través de la página `http://lml.ls.fi.upm.es/entrega`. Es obligatorio entregar fichero Java con codificación de caracteres UTF-8 (configura para ello tu editor o entorno de desarrollo favorito).

Nota: cuidado al copiar y pegar código de este PDF, puede que el resultado no sea el esperado en el editor.

Índice

1. Creación de threads en Java

Con lo visto en clase y la documentación sobre concurrencia en los tutoriales de Java (`http://docs.oracle.com/javase/tutorial/essential/concurrency/`, se pide escribir un programa concurrente en Java que arranque N threads y termine cuando los N threads terminen. Todos los threads deben realizar el mismo trabajo: imprimir una línea que los identifique y distinga (no se permite el uso de `Thread.currentThread()` ni los métodos `getId()` o `toString()` o `getName()` de la clase `Thread`), dormir durante T milisegundos y terminar imprimiendo una línea que los identifique y distinga. El *thread* principal, además de poner en marcha todos los procesos, debe imprimir una línea avisando de que todos los threads han terminado una vez que lo hayan hecho.

Es un ejercicio muy sencillo que debe servir para jugar con el concepto de proceso intentando distinguir lo que cada proceso hace y el momento en el que lo hace. Además, se podrá observar cómo cada ejecución lleva a resultados diferentes. Se sugiere jugar con los valores N y T e incluso hacer que T sea distinto para cada proceso.

Material a entregar

El fichero fuente a entregar debe llamarse `CC_01_Threads.java`

2. Provocar una condición de carrera

Este ejercicio consiste en escribir un programa concurrente en el que múltiples threads compartan y modifiquen una variable de tipo int de forma que el resultado final de la variable una vez que los threads terminan no sea el valor esperado. Seamos más concretos. Tendremos dos tipos de procesos, decrementadores e incrementadores que realizan N decrementos e incrementos, respectivamente, sobre una misma variable (`n`) de tipo `int` inicializada a 0. El programa concurrente pondrá en marcha M procesos de cada tipo y una vez que todos los threads han terminado imprimirá el valor de la variable compartida.

El valor final de la variable debería ser 0 ya que se habrán producido $M \times N$ decrementos (`n--`) y $M \times N$ incrementos (`n++`), sin embargo, si dos operaciones (tanto de decremento como de incremento) se realizan a la vez el resultado puede no ser el esperado (por ejemplo, dos incrementos podrían terminar por no incrementar la variable en 2).

El alumno no debería realizar la entrega hasta que no vea que el valor final de la variable puede ser distinto de 0 (aunque esto no garantiza que haya una condición de carrera).

Material a entregar

El fichero fuente a entregar debe llamarse `CC_02_Carrera.java`.

3. Garantizar exclusión mutua con espera activa

Este ejercicio consiste en evitar la condición de carrera que se produjo en el ejercicio anterior. Para ello supondremos la existencia de **sólo dos procesos**, que simultáneamente ejecutan sendos bucles de N pasos incrementando y decrementando, respectivamente, en cada paso una variable compartida (la operación de incremento y la de decremento sobre esa misma variable compartida son secciones críticas). El objetivo es evitar que mientras un proceso modifica la variable el otro haga lo mismo (propiedad que se denomina exclusión mutua: no puede haber dos procesos modificando simultáneamente esa variable) y el objetivo es hacerlo utilizando sólo nuevas variables y "espera activa" (en otras palabras, está prohibido utilizar métodos synchronized, semáforos o cualquier otro mecanismo de concurrencia).

Material a entregar

El fichero fuente a entregar debe llamarse `CC_03_MutexEA.java`.

Material de apoyo

Se ofrece un **intento fallido** para asegurar la exclusión mutua en la ejecución de las secciones críticas en el fichero `CC_03_MutexEA.todo.java`:

```
// Exclusión mutua con espera activa.
//
// Intentar garantizar la exclusión mutua en sc_inc y sc_dec sin
// utilizar más mecanismo de concurrencia que el de la espera activa
// (nuevas variables y bucles).
//
// Las propiedades que deberán cumplirse:
// - Garantía mutual exclusión (exclusión mútua): nunca hay dos
//   procesos ejecutando secciones críticas de forma simultánea.
// - Ausencia de deadlock (interbloqueo): los procesos no quedan
//   "atrapados" para siempre.
// - Ausencia de starvation (inanición): si un proceso quiere acceder
//   a su sección crítica entonces es seguro que alguna vez lo hace.
// - Ausencia de esperas innecesarias: si un proceso queire acceder a
//   su sección crítica y ningún otro proceso está accediendo ni
//   quiere acceder entonces el primero puede acceder.
//
// Ideas:
// - Una variable booleana en_sc que indica que algún proceso está
//   ejecutando en la sección crítica?
// - Una variable booleana turno?
// - Dos variables booleanas en_sc_inc y en_sc_dec que indican que un
//   determinado proceso (el incrementador o el decrementador) está
//   ejecutando su sección crítica?
// - Combinaciones?

class CC_03_MutexEA {
    static final int N_PASOS = 10000;
```

```
// Generador de números aleatorios para simular tiempos de
// ejecución
// static final java.util.Random RNG = new java.util.Random(0);

// Variable compartida
volatile static int n = 0;

// Variables para asegurar exclusión mutua
volatile static boolean en_sc = false;

// Sección no crítica
static void no_sc() {
   // System.out.println("No SC");
   // try {
   //    // No más de 2ms
   //    Thread.sleep(RNG.nextInt(3));
   // }
   // catch (Exception e) {
   //    e.printStackTrace();
   // }
}

// Secciones críticas
static void sc_inc() {
   // System.out.println("Incrementando");
   n++;
}

static void sc_dec() {
   // System.out.println("Decrementando");
   n--;
}

// La labor del proceso incrementador es ejecutar no_sc() y luego
// sc_inc() durante N_PASOS asegurando exclusión mutua sobre
// sc_inc().
static class Incrementador extends Thread {
   public void run () {
      for (int i = 0; i < N_PASOS; i++) {
         // Sección no crítica
         no_sc();

         // Protocolo de acceso a la sección crítica
         while (en_sc) {}
         en_sc = true;

         // Sección crítica
         sc_inc();

         // Protocolo de salida de la sección crítica
         en_sc = false;
      }
   }
```

```
   }

   // La labor del proceso incrementador es ejecutar no_sc() y luego
   // sc_dec() durante N_PASOS asegurando exclusión mutua sobre
   // sc_dec().
   static class Decrementador extends Thread {
      public void run () {
         for (int i = 0; i < N_PASOS; i++) {
            // Sección no crítica
            no_sc();

            // Protocolo de acceso a la sección crítica
            while (en_sc) {}
            en_sc = true;

            // Sección crítica
            sc_dec();

            // Protocolo de salida de la sección crítica
            en_sc = false;
         }
      }
   }

   public static final void main(final String[] args)
      throws InterruptedException
   {
      // Creamos las tareas
      Thread t1 = new Incrementador();
      Thread t2 = new Decrementador();

      // Las ponemos en marcha
      t1.start();
      t2.start();

      // Esperamos a que terminen
      t1.join();
      t2.join();

      // Simplemente se muestra el valor final de la variable:
      System.out.println(n);
   }
}
```

4. Garantizar exclusión mutua con semáforos

Este ejercicio, al igual que el anterior, consiste en evitar una condición de carrera. En esta ocasión tenemos el mismo número de procesos incrementadores que decrementadores que incrementan y decrementan, respectivamente, en un mismo número de pasos una variable compartida. El objetivo es asegurar la exclusión mutua en la ejecución de los incrementos y decrementos de la variable y el objetivo es hacerlo utilizando exclusivamente un semáforo de la clase `es.upm.babel.cclib.Semaphore` (está prohibido utilizar cualquier otro mecanismo de concurrencia). La librería de concurrencia `cclib.jar` puede descargarse de la página web de la asignatura.

Material a entregar

El fichero fuente a entregar debe llamarse `CC_04_MutexSem.java`.

Material de apoyo

Se ofrece el fichero `CC_04_MutexSem.todo.java` con un esqueleto que debe respetarse y que muestra una condición de carrera:

```
import es.upm.babel.cclib.Semaphore;

class CC_04_MutexSem {
   private static int N_THREADS = 2;
   private static int N_PASOS = 1000000;

   static class Contador {
      private volatile int n;
      public Contador() {
         this.n = 0;
      }
      public int valorContador() {
         return this.n;
      }
      public void inc () {
         this.n++;
      }
      public void dec () {
         this.n--;
      }
   }

   static class Incrementador extends Thread {
      private Contador cont;
      public Incrementador (Contador c) {
         this.cont = c;
      }
      public void run() {
         for (int i = 0; i < N_PASOS; i++) {
            this.cont.inc();
         }
```

```
    }
  }

  static class Decrementador extends Thread {
    private Contador cont;
    public Decrementador (Contador c) {
      this.cont = c;
    }
    public void run() {
      for (int i = 0; i < N_PASOS; i++) {
        this.cont.dec();
      }
    }
  }

  public static void main(String args[])
  {
    // Creación del objeto compartido
    Contador cont = new Contador();

    // Creación de los arrays que contendrán los threads
    Incrementador[] tInc =
      new Incrementador[N_THREADS];
    Decrementador[] tDec =
      new Decrementador[N_THREADS];

    // Creacion de los objetos threads
    for (int i = 0; i < N_THREADS; i++) {
      tInc[i] = new Incrementador(cont);
      tDec[i] = new Decrementador(cont);
    }

    // Lanzamiento de los threads
    for (int i = 0; i < N_THREADS; i++) {
      tInc[i].start();
      tDec[i].start();
    }

    // Espera hasta la terminacion de los threads
    try {
      for (int i = 0; i < N_THREADS; i++) {
        tInc[i].join();
        tDec[i].join();
      }
    } catch (Exception ex) {
      ex.printStackTrace();
      System.exit (-1);
    }

    // Simplemente se muestra el valor final de la variable:
    System.out.println(cont.valorContador());
    System.exit (0);
  }
```

```
}
```

5. Almacén de un dato con semáforos

En este caso nos enfrentamos a un típico programa de concurrencia: productores-consumidores. Existen procesos de dos tipos diferentes:

- Productores: su hilo de ejecución consiste, repetidamente, en crear un producto (ver la clase `es.upm.babel.cclib.Producto`) y hacerlo llegar a uno de los consumidores.
- Consumidores: su hilo de ejecución consiste, repetidamente, recoger productos producidos por los productores y consumirlos.

Las clases que implementan ambos threads forman parte de la librería CCLib:
`es.upm.babel.cclib.Productor` y `es.upm.babel.cclib.Consumidor`.

La comunicación entre productores y consumidores se realizará a través de un "almacén" compartido por todos los procesos. Dicho objeto respetará la interfaz `es.upm.babel.cclib.Almacen`:

```
package es.upm.babel.cclib;

/**
 * Interfaz para almacén concurrente.
 */
public interface Almacen {
   /**
    * Almacena (como último) un producto en el almacén. Si no hay
    * hueco el proceso que ejecute el método bloqueará hasta que lo
    * haya.
    */
   public void almacenar(Producto producto);

   /**
    * Extrae el primer producto disponible. Si no hay productos el
    * proceso que ejecute el método bloqueará hasta que se almacene un
    * dato.
    */
   public Producto extraer();
}
```

Se pide implementar sólo con semáforos una clase que siga dicha interfaz. Sólo puede haber almacenado como máximo un producto, si un proceso quiere almacenar debe esperar hasta que no haya un producto y si un proceso quiera extraer espere hasta que haya un producto. Téngase en cuenta además los posible problemas de no asegurar la exclusión mutua en el acceso a los atributos compartidos.

Material a entregar

El fichero fuente a entregar debe llamarse `Almacen1.java`.

Material de apoyo

Se ofrece un esqueleto de código que el alumno debe completar en el fichero `Almacen1.todo.java`:

```
import es.upm.babel.cclib.Producto;
import es.upm.babel.cclib.Almacen;

// TODO: importar la clase de los semáforos.

/**
 * Implementación de la clase Almacen que permite el almacenamiento
 * de producto y el uso simultáneo del almacen por varios threads.
 */
class Almacen1 implements Almacen {
   // Producto a almacenar: null representa que no hay producto
   private Producto almacenado = null;

   // TODO: declaración e inicialización de los semáforos
   // necesarios

   public Almacen1() {
   }

   public void almacenar(Producto producto) {
      // TODO: protocolo de acceso a la sección crítica y código de
      // sincronización para poder almacenar.

      // Sección crítica
      almacenado = producto;

      // TODO: protocolo de salida de la sección crítica y código de
      // sincronización para poder extraer.
   }

   public Producto extraer() {
      Producto result;

      // TODO: protocolo de acceso a la sección crítica y código de
      // sincronización para poder extraer.

      // Sección crítica
      result = almacenado;
      almacenado = null;

      // TODO: protocolo de salida de la sección crítica y código de
      // sincronización para poder almacenar.

      return result;
   }
}
```

El programa principal `CC_05_P1CSem.java` es el siguiente:

```
import es.upm.babel.cclib.Producto;
import es.upm.babel.cclib.Almacen;
import es.upm.babel.cclib.Productor;
import es.upm.babel.cclib.Consumidor;
import es.upm.babel.cclib.Consumo;
import es.upm.babel.cclib.Fabrica;

/**
 * Programa concurrente para productor-buffer-consumidor con almacen
 * de tamaño 1 implementado con semáforos (Almacen1).
 */
class CC_05_P1CSem {
    public static final void main(final String[] args)
       throws InterruptedException
    {
        // Número de productores y consumidores
        final int N_PRODS = 2;
        final int N_CONSS = 2;

        Consumo.establecerTiempoMedioCons(100);
        Fabrica.establecerTiempoMedioProd(100);

        // Almacen compartido
        Almacen almac = new Almacen1();

        // Declaración de los arrays de productores y consumidores
        Productor[] productores;
        Consumidor[] consumidores;

        // Creación de los arrays
        productores = new Productor[N_PRODS];
        consumidores = new Consumidor[N_CONSS];

        // Creación de los productores
        for (int i = 0; i < N_PRODS; i++) {
            productores[i] = new Productor(almac);
        }

        // Creación de los consumidores
        for (int i = 0; i < N_CONSS; i++) {
            consumidores[i] = new Consumidor(almac);
        }

        // Lanzamiento de los productores
        for (int i = 0; i < N_PRODS; i++) {
            productores[i].start();
        }

        // Lanzamiento de los consumidores
        for (int i = 0; i < N_CONSS; i++) {
            consumidores[i].start();
        }
```

```
        // Espera hasta la terminación de los procesos
        try {
            for (int i = 0; i < N_PRODS; i++) {
                productores[i].join();
            }
            for (int i = 0; i < N_CONSS; i++) {
                consumidores[i].join();
            }
        } catch (Exception ex) {
            ex.printStackTrace();
            System.exit (-1);
        }
    }
}
```

Para valorar si el problema está bien resuelto, os recordamos que el objetivo es asegurar

1. que todos los productos producidos acaban por ser consumidos,
2. que no se consume un producto dos veces y
3. que no se consume ningún producto no válido (null, por ejemplo).

Recomendación: jugar con los valores de número de productores y número de consumidores y observar con atención las trazas del programa.

6. Almacén de varios datos con semáforos

Este ejercicio es una variación del problema anterior. En esta ocasión, el almacén a implementar tiene una capacidad de hasta N productos, lo que permite a los productores seguir trabajando aunque los consumidores se vuelvan, momentáneamente, lentos.

Material a entregar

El fichero fuente a entregar debe llamarse `AlmacenN.java`.

Material de apoyo

Se ofrece un esqueleto de código que el alumno debe completar en el fichero `AlmacenN.todo.java`:

```
import es.upm.babel.cclib.Producto;
import es.upm.babel.cclib.Almacen;

// TODO: importar la clase de los semáforos.

/**
 * Implementación de la clase Almacen que permite el almacenamiento
 * FIFO de hasta un determinado número de productos y el uso
 * simultáneo del almacén por varios threads.
 */
class AlmacenN implements Almacen {
   private int capacidad = 0;
   private Producto[] almacenado = null;
   private int nDatos = 0;
   private int aExtraer = 0;
   private int aInsertar = 0;

   // TODO: declaración de los semáforos necesarios

   public AlmacenN(int n) {
      capacidad = n;
      almacenado = new Producto[capacidad];
      nDatos = 0;
      aExtraer = 0;
      aInsertar = 0;

      // TODO: inicialización de los semáforos
   }

   public void almacenar(Producto producto) {
      // TODO: protocolo de acceso a la sección crítica y código de
      // sincronización para poder almacenar.

      // Sección crítica
      almacenado[aInsertar] = producto;
```

```
      nDatos++;
      aInsertar++;
      aInsertar %= capacidad;

      // TODO: protocolo de salida de la sección crítica y código de
      // sincronización para poder extraer.
   }

   public Producto extraer() {
      Producto result;

      // TODO: protocolo de acceso a la sección crítica y código de
      // sincronización para poder extraer.

      // Sección crítica
      result = almacenado[aExtraer];
      almacenado[aExtraer] = null;
      nDatos--;
      aExtraer++;
      aExtraer %= capacidad;

      // TODO: protocolo de salida de la sección crítica y código de
      // sincronización para poder almacenar.

      return result;
   }
}
```

y el programa principal `CC_06_PNSem.java` es el siguiente:

```
import es.upm.babel.cclib.Producto;
import es.upm.babel.cclib.Almacen;
import es.upm.babel.cclib.Productor;
import es.upm.babel.cclib.Consumidor;

/**
 * Programa concurrente para productor-buffer-consumidor con almacen
 * de capacidad N implementado con semáforos (AlmacenN).
 */
class CC_06_PNCSem {
   public static final void main(final String[] args)
      throws InterruptedException
   {
      // Capacidad del buffer
      final int CAPACIDAD = 10;

      // Número de productores y consumidores
      final int N_PRODS = 2;
      final int N_CONSS = 2;

      // Almacen compartido
      Almacen almac = new AlmacenN(CAPACIDAD);

      // Declaración de los arrays de productores y consumidores
```

```
      Productor[] productores;
      Consumidor[] consumidores;

      // Creación de los arrays
      productores = new Productor[N_PRODS];
      consumidores = new Consumidor[N_CONSS];
      // Creación de los productores
      for (int i = 0; i < N_PRODS; i++) {
         productores[i] = new Productor(almac);
      }

      // Creación de los consumidores
      for (int i = 0; i < N_CONSS; i++) {
         consumidores[i] = new Consumidor(almac);
      }

      // Lanzamiento de los productores
      for (int i = 0; i < N_PRODS; i++) {
         productores[i].start();
      }

      // Lanzamiento de los consumidores
      for (int i = 0; i < N_CONSS; i++) {
         consumidores[i].start();
      }

      // Espera hasta la terminación de los procesos
      try {
         for (int i = 0; i < N_PRODS; i++) {
            productores[i].join();
         }
         for (int i = 0; i < N_CONSS; i++) {
            consumidores[i].join();
         }
      } catch (Exception ex) {
         ex.printStackTrace();
         System.exit (-1);
      }
   }
}
```

7. Especificación de un recurso compartido

El ejercicio consiste en **elaborar la especificación formal del recurso compartido ControlAccesoPuente**. El recurso compartido forma parte de un programa concurrente que gestiona los accesos y salidas de coches de un puente de un solo carril. El puente tiene dos entradas y dos salidas. Los coches que entran por la entrada sur salen por la salida norte y viceversa. En cada entrada existe un detector y una barrera. En cada salida existe un detector. El sistema de detección y barreras tienen el interfaz de control `Puente.java`[1]:

```
/**
 * Interfaz de control de acceso al puente de un solo sentido.
 *
 * Hay dos accesos de entrada al puente, uno al norte y otro al sur.
 *
 * Hay dos salidas del puente, una al norte y otra al sur.
 *
 * En las entradas y en las salidas hay detectores. Para detectar que
 * un vehículo ha llegado a un detector se usan los métodos detectar.
 *
 * Las entradas están controladas con barreras. Para abrir una barrera
 * se utiliza el método abrir.
 */
public class Puente {
   // TODO: introducir atributos para la simulación (generadores
   // aleatorios, etc.)

   /**
    * Enumerado con los identificadores de las entradas.
    */
   static public enum Entrada { N, S }

   /**
    * Enumerado con los identificadores de las salidas.
    */
   static public enum Salida { N, S }

   /**
    * El thread que invoque la operación detectar queda bloqueado
    * hasta que un coche llegue el detector indicado, en ese momento
    * el thread termina de ejecutar el método.
    */
   static public void detectar(Entrada e) {
      // TODO: elaborar el código de simulación
   }

   /**
    * El thread que invoque la operación detectar queda bloqueado
    * hasta que un coche llegue el detector indicado, en ese momento
    * el thread termina de ejecutar el método.
    */
```

[1] Los comentarios tipo TODO no significan que el alumno tenga que implementar nada, sólo son apuntes para disponer eventualmente de una simulación.

```
   static public void detectar(Salida s) {
      // TODO: elaborar el código de simulación
   }

   /**
    * Al invocar la operación se abrirá la barrera indicada, se
    * permite la entrada de un coche al puente y se cierra la barrera
    * indicada. El tiempo que tarda en ejecutarse el método abrir
    * coincide con el tiempo que tarda en realizarse toda la actividad
    * (abrir-pasar-cerrar).
    */
   static public void abrir(Entrada e) {
      // TODO: elaborar el código de simulación
   }
}
```

El objetivo del programa concurrente es controlar la entrada y salida de vehículos de tal forma que jamás haya en el puente dos coches que pretenda dirigirse en sentido contrario. Se ha decidido un diseño en el que existe un proceso por cada entrada y un proceso por cada salida. Los procesos comparten un recurso compartido que les sirve para comunicarse.

A continuación se muestra el interfaz del recurso compartido que hay que especificar (`ControlAccesoPuente.java`):

```
public class ControlAccesoPuente {

   public ControlAccesoPuente() {
   }

   /**
    * Incrementa el número de coches en el puente que han entrado por
    * la entrada e. Si los coches en el puente van en el sentido
    * opuesto entonces el proceso que lo invoque debe bloquear.
    */
   public void solicitarEntrada (Puente.Entrada e) {
   }

   /**
    * Decrementa el número de coches en el puente.
    */
   public void avisarSalida (Puente.Salida s) {
   }
}
```

y el programa concurrente `CC_07_Puente.java`:

```
/**
 * Programa concurrente para el control del acceso al puente de un
 * solo sentido.
 */
class CC_07_Puente {

   static private class ControlEntrada extends Thread {
      private Puente.Entrada e;
```

```
      private ControlAccesoPuente cap;

      public ControlEntrada (Puente.Entrada e,
                             ControlAccesoPuente cap) {
         this.e = e;
         this.cap = cap;
      }

      public void run() {
         while (true) {
            Puente.detectar(this.e);
            cap.solicitarEntrada(this.e);
            Puente.abrir(this.e);
         }
      }
   }

   static private class AvisoSalida extends Thread {
      private Puente.Salida s;
      private ControlAccesoPuente cap;

      public AvisoSalida (Puente.Salida s,
                          ControlAccesoPuente cap) {
         this.s = s;
         this.cap = cap;
      }

      public void run() {
         while (true) {
            Puente.detectar(this.s);
            cap.avisarSalida(this.s);
         }
      }
   }

   public static final void main(final String[] args)
      throws InterruptedException
   {
      ControlAccesoPuente cap;
      ControlEntrada ceN, ceS;
      AvisoSalida asN, asS;

      cap = new ControlAccesoPuente();
      ceN = new ControlEntrada(Puente.Entrada.N,cap);
      ceS = new ControlEntrada(Puente.Entrada.S,cap);
      asN = new AvisoSalida(Puente.Salida.N,cap);
      asS = new AvisoSalida(Puente.Salida.S,cap);

      ceN.start();
      ceS.start();
      asN.start();
      asS.start();
   }
```

```
}
```

8. Multibuffer con métodos synchronized

El *MultiBuffer* es una variación del problema del búffer compartido en el que productores y consumidores pueden insertar o extraer secuencias de elementos de longitud arbitraria, lo cual lo convierte en un ejemplo más realista. A diferencia de versiones más sencillas, este es un ejercicio de programación difícil si sólo se dispone de mecanismos de sincronización de bajo nivel (p.ej. semáforos).

Por ello, os pedimos que lo implementéis en Java traduciendo la siguiente especificación de recurso a una clase usando métodos *synchronized* y el mecanismo `wait()/notifyAll()`.

C-TAD MultiBuffer

OPERACIONES
ACCIÓN Poner: *Tipo_Secuencia[e]*
ACCIÓN Tomar: $\mathbb{N}$*[e]* $\times$ *Tipo_Secuencia[s]*

SEMÁNTICA
DOMINIO:
TIPO: *MultiBuffer* = Secuencia*(Tipo_Dato)*
Tipo_Secuencia = MultiBuffer
INVARIANTE: Longitud(self) $\leq MAX$
DONDE: $MAX = ...$
INICIAL: self $= \langle\rangle$

PRE: $n \leq \lfloor MAX/2 \rfloor$
***CPRE*:** *Hay suficientes elementos en el multibuffer*
CPRE: Longitud(self) $\geq n$
Tomar(n, s)
***POST*:** *Retiramos elementos*
POST: $n =$ Longitud$(s) \wedge$ self$^{pre} = s$ + self

PRE: Longitud$(s) \leq \lfloor MAX/2 \rfloor$
***CPRE*:** *Hay sitio en el buffer para dejar la secuencia*
CPRE: Longitud(self $+ s) \leq MAX$
Poner(s)
***POST*:** *Añadimos una secuencia al buffer*
POST: self $=$ self^{pre} + s^{pre}
Observad que la especificación contiene también precondiciones para evitar situaciones de interbloqueo.

Material a entregar

El fichero fuente a entregar debe llamarse `MultiAlmacenSync.java`.

Material de apoyo

Se ofrece un esqueleto de código que el alumno debe completar en el fichero `MultiAlmacenSync.todo.java`:

```
import es.upm.babel.cclib.Producto;
import es.upm.babel.cclib.MultiAlmacen;

class MultiAlmacenSync implements MultiAlmacen {
    private int capacidad = 0;
    private Producto almacenado[] = null;
    private int aExtraer = 0;
    private int aInsertar = 0;
    private int nDatos = 0;

   // TODO: declaración de atributos extras necesarios

   // Para evitar la construcción de almacenes sin inicializar la
   // capacidad
   private MultiAlmacenSync() {
   }

   public MultiAlmacenSync(int n) {
      almacenado = new Producto[n];
      aExtraer = 0;
      aInsertar = 0;
      capacidad = n;
      nDatos = 0;

      // TODO: inicialización de otros atributos
   }

   private int nDatos() {
         return nDatos;
   }

   private int nHuecos() {
      return capacidad - nDatos;
   }

   synchronized public void almacenar(Producto[] productos) {
      // TODO: implementación de código de bloqueo para sincronización
      // condicional

      // Sección crítica
      for (int i = 0; i < productos.length; i++) {
         almacenado[aInsertar] = productos[i];
         nDatos++;
         aInsertar++;
         aInsertar %= capacidad;
      }

      // TODO: implementación de código de desbloqueo para
      // sincronización condicional
```

```
   }

   synchronized public Producto[] extraer(int n) {
      Producto[] result = new Producto[n];

      // TODO: implementación de código de bloqueo para sincronización
      // condicional

      // Sección crítica
      for (int i = 0; i < result.length; i++) {
         result[i] = almacenado[aExtraer];
         almacenado[aExtraer] = null;
         nDatos--;
         aExtraer++;
         aExtraer %= capacidad;
      }

      // TODO: implementación de código de desbloqueo para
      // sincronización condicional

      return result;
   }
}
```

El programa principal para probar este ejercicio es el `CC_08_PmultiCSync.java`:

```
import es.upm.babel.cclib.MultiAlmacen;
import es.upm.babel.cclib.MultiProductor;
import es.upm.babel.cclib.MultiConsumidor;

/**
 * Programa concurrente para productor-buffer-consumidor con multialmacen
 * de capacidad N implementado con métodos synchronized (MultiAlmacenSync).
 */
class CC_08_PmultiCSync {
    public static final void main(final String[] args)
        throws InterruptedException
    {
        // Capacidad del multialmacen
        final int N = 10;

        // Número de productores y consumidores
        final int N_PRODS = 2;
        final int N_CONSS = 2;

        // Máxima cantidad de productos por paquete para producir y
        // consumir
        final int MAX_PROD = N / 2;
        final int MAX_CONS = N / 2;

        // Almacen compartido
        MultiAlmacen almac = new MultiAlmacenSync(N);

        // Declaración de los arrays de productores y consumidores
```

```
        MultiProductor[] productores;
        MultiConsumidor[] consumidores;

        // Creación de los arrays
        productores = new MultiProductor[N_PRODS];
        consumidores = new MultiConsumidor[N_CONSS];

        // Creación de los productores
        for (int i = 0; i < N_PRODS; i++) {
            productores[i] = new MultiProductor(almac,MAX_PROD);
        }

        // Creación de los consumidores
        for (int i = 0; i < N_CONSS; i++) {
            consumidores[i] = new MultiConsumidor(almac,MAX_CONS);
        }

        // Lanzamiento de los productores
        for (int i = 0; i < N_PRODS; i++) {
            productores[i].start();
        }

        // Lanzamiento de los consumidores
        for (int i = 0; i < N_CONSS; i++) {
            consumidores[i].start();
        }

        // Espera hasta la terminación de los procesos
        try {
            for (int i = 0; i < N_PRODS; i++) {
                productores[i].join();
            }
            for (int i = 0; i < N_CONSS; i++) {
                consumidores[i].join();
            }
        } catch (Exception ex) {
            ex.printStackTrace();
            System.exit (-1);
        }
    }
}
```

9. Multibuffer con monitores

En esta versión del ejercicio, la tarea consiste en resolver el problema anterior usando las clases `es.upm.babel.cclib.Monitor` y `es.upm.babel.cclib.Monitor.Cond`.

Material a entregar

El fichero fuente a entregar debe llamarse `MultiAlmacenMon.java`.

Material de apoyo

Se ofrece un esqueleto de código que el alumno debe completar en el fichero `MultiAlmacenMon.todo.java`:

```
import es.upm.babel.cclib.Producto;
import es.upm.babel.cclib.MultiAlmacen;

// importar la librería de monitores


class MultiAlmacenMon implements MultiAlmacen {
    private int capacidad = 0;
    private Producto almacenado[] = null;
    private int aExtraer = 0;
    private int aInsertar = 0;
    private int nDatos = 0;

   // TODO: declaración de atributos extras necesarios
   // para exclusión mutua y sincronización por condición


   // Para evitar la construcción de almacenes sin inicializar la
   // capacidad
   private MultiAlmacenMon() {
   }

   public MultiAlmacenMon(int n) {
      almacenado = new Producto[n];
      aExtraer = 0;
      aInsertar = 0;
      capacidad = n;
      nDatos = 0;

      // TODO: inicialización de otros atributos
   }

   private int nDatos() {
         return nDatos;
   }

   private int nHuecos() {
```

```
      return capacidad - nDatos;
   }

   public void almacenar(Producto[] productos) {

      // TODO: implementación de código de bloqueo para
      // exclusión muytua y sincronización condicional

      // Sección crítica
      for (int i = 0; i < productos.length; i++) {
         almacenado[aInsertar] = productos[i];
         nDatos++;
         aInsertar++;
         aInsertar %= capacidad;
      }

      // TODO: implementación de código de desbloqueo para
      // sincronización condicional y liberación de la exclusión mutua
   }

   public Producto[] extraer(int n) {
      Producto[] result = new Producto[n];

      // TODO: implementación de código de bloqueo para exclusión
      // mutua y sincronización condicional

      // Sección crítica
      for (int i = 0; i < result.length; i++) {
         result[i] = almacenado[aExtraer];
         almacenado[aExtraer] = null;
         nDatos--;
         aExtraer++;
         aExtraer %= capacidad;
      }

      // TODO: implementación de código de desbloqueo para
      // sincronización condicional y liberación de la exclusión mutua

      return result;
   }
}
```

El programa principal para probar este ejercicio es el `CC_09_PmultiCMon.java`:

```
import es.upm.babel.cclib.MultiAlmacen;
import es.upm.babel.cclib.MultiProductor;
import es.upm.babel.cclib.MultiConsumidor;

/**
 * Programa concurrente para productor-buffer-consumidor con multialmacen
 * de capacidad N implementado con monitores (MultiAlmacenMon).
 */
class CC_09_PmultiCMon {
    public static final void main(final String[] args)
```

```
        throws InterruptedException {

        // Capacidad del multialmacen
        final int N = 10;

        // Número de productores y consumidores
        final int N_PRODS = 2;
        final int N_CONSS = 2;

        // Máxima cantidad de productos por paquete para producir y consumir
        final int MAX_PROD = N / 2;
        final int MAX_CONS = N / 2;

        // Almacen compartido
        MultiAlmacen almac = new MultiAlmacenMon(N);

        // Declaración de los arrays de productores y consumidores
        MultiProductor[] productores;
        MultiConsumidor[] consumidores;

        // Creación de los arrays
        productores = new MultiProductor[N_PRODS];
        consumidores = new MultiConsumidor[N_CONSS];

        // Creación de los productores
        for (int i = 0; i < N_PRODS; i++) {
            productores[i] = new MultiProductor(almac,MAX_PROD);
        }

        // Creación de los consumidores
        for (int i = 0; i < N_CONSS; i++) {
            consumidores[i] = new MultiConsumidor(almac,MAX_CONS);
        }

        // Lanzamiento de los productores
        for (int i = 0; i < N_PRODS; i++) {
            productores[i].start();
        }

        // Lanzamiento de los consumidores
        for (int i = 0; i < N_CONSS; i++) {
            consumidores[i].start();
        }

        // Espera hasta la terminación de los procesos
        try {
            for (int i = 0; i < N_PRODS; i++) {
                productores[i].join();
            }
            for (int i = 0; i < N_CONSS; i++) {
                consumidores[i].join();
            }
        } catch (Exception ex) {
```

```
            ex.printStackTrace();
            System.exit (-1);
        }
    }
}
```

10. Multibuffer con paso de mensajes

Se trata de resolver el problema anterior con paso de mensajes usando la librería JCSP[2].

Material a entregar

El fichero fuente a entregar debe llamarse `MultiAlmacenJCSP.java`.

Material de apoyo

Se ofrece un esqueleto de código que el alumno debe completar en el fichero `MultiAlmacenJCSP.todo.java`:

```
import es.upm.babel.cclib.Producto;
import es.upm.babel.cclib.MultiAlmacen;

// importamos la librería JCSP
import org.jcsp.lang.*;

class MultiAlmacenJCSP implements MultiAlmacen, CSProcess {

    // Canales para enviar y recibir peticiones al/del servidor
    private final Any2OneChannel chAlmacenar = Channel.any2one();
    private final Any2OneChannel chExtraer = Channel.any2one();
    private int TAM;

    // Para evitar la construcción de almacenes sin inicializar la
    // capacidad
    private MultiAlmacenJCSP() {
    }

    public MultiAlmacenJCSP(int n) {
        this.TAM = n;

        // COMPLETAR: inicialización de otros atributos

    }

    public void almacenar(Producto[] productos) {

        // COMPLETAR: comunicación con el servidor

    }

    public Producto[] extraer(int n) {
        Producto[] result = new Producto[n];

        // COMPLETAR: comunicación con el servidor
```

[2] `http://www.cs.kent.ac.uk/projects/ofa/jcsp/`

```
        return result;
    }


    // código del servidor
    private static final int ALMACENAR = 0;
    private static final int EXTRAER = 1;
    public void run() {
        // COMPLETAR: declaración de canales y estructuras auxiliares

        Guard[] entradas = {
            chAlmacenar.in(),
            chExtraer.in()
        };
        Alternative servicios = new Alternative(entradas);
        int choice = 0;

        while (true) {
            try {
                choice = servicios.fairSelect();
            } catch (ProcessInterruptedException e){}
            switch(choice){
            case ALMACENAR:

                // COMPLETAR: tratamiento de la petición

                break;
            case EXTRAER:

                // COMPLETAR: tratamiento de la petición

                break;
            }

            // COMPLETAR: atención de peticiones pendientes
        }
    }
}
```

El programa principal para probar este ejercicio es el `CC_10_PmultiJCSP.java`:

```
import es.upm.babel.cclib.MultiAlmacen;
import es.upm.babel.cclib.MultiProductor;
import es.upm.babel.cclib.MultiConsumidor;
import org.jcsp.lang.*;

/**
 * Programa concurrente para productor-buffer-consumidor con multialmacen
 * de capacidad N implementado con paso de mensajes (MultiAlmacenJCSP).
 */
class CC_10_PmultiCJCSP {
    public static final void main(final String[] args)
        throws InterruptedException {
```

```
        // Capacidad del multialmacen
        final int N = 10;

        // Número de productores y consumidores
        final int N_PRODS = 2;
        final int N_CONSS = 2;

        // Máxima cantidad de productos por paquete para producir y consumir
        final int MAX_PROD = N / 2;
        final int MAX_CONS = N / 2;

        // Almacen compartido
        MultiAlmacenJCSP almac = new MultiAlmacenJCSP(N);
        // OJO!!
        ProcessManager m_almac = new ProcessManager(almac);

        // Lanzamos el servidor del almacén
        m_almac.start();

        // Declaración de los arrays de productores y consumidores
        MultiProductor[] productores;
        MultiConsumidor[] consumidores;

        // Creación de los arrays
        productores = new MultiProductor[N_PRODS];
        consumidores = new MultiConsumidor[N_CONSS];

        // Creación de los productores
        for (int i = 0; i < N_PRODS; i++) {
            productores[i] = new MultiProductor(almac,MAX_PROD);
        }

        // Creación de los consumidores
        for (int i = 0; i < N_CONSS; i++) {
            consumidores[i] = new MultiConsumidor(almac,MAX_CONS);
        }

        // Lanzamiento de los productores
        for (int i = 0; i < N_PRODS; i++) {
            productores[i].start();
        }

        // Lanzamiento de los consumidores
        for (int i = 0; i < N_CONSS; i++) {
            consumidores[i].start();
        }

        // Espera hasta la terminación de los clientes
        try {
            for (int i = 0; i < N_PRODS; i++) {
                productores[i].join();
            }
            for (int i = 0; i < N_CONSS; i++) {
```

```
                consumidores[i].join();
            }
        } catch (Exception ex) {
            ex.printStackTrace();
            System.exit (-1);
        }
    }
}
```