BOLETÍN DE ACTIVIDADES

CUESTION 1. Dadas las siguientes afirmaciones, modifique en su caso lo que sea necesario para que las afirmaciones sean **CIERTAS.** *Nota*: Puede haber afirmaciones que ya sean inicialmente ciertas.

1.	Dos hilos de dos procesos distintos pueden comunicarse entre sí utilizando memoria compartida.
2.	Se produce una condición de carrera cuando se realiza una modificación de la memoria compartida que da lugar a un estado consistente.
3.	En el problema Lectores-Escritores se requiere acceso en exclusión mutua solamente para los escritores entre sí.
4.	La sincronización condicional implica suspender a un hilo durante un tiempo máximo especificado en la condición.
5.	La sección crítica es un fragmento de código que puede provocar condiciones de carrera.
6.	Para dar solución a las condiciones de carrera, basta con emplear un protocolo de entrada que garantice las propiedades de exclusión mutua, espera limitada y seguridad.
7.	Un `lock` es un objeto que, cuando un hilo H1 utiliza su operación "`cerrar lock`", si estaba ya abierto por H1, entonces no tiene efecto.
8.	El problema de la sección crítica aparece cuando varios hilos comparten un objeto inmutable (constante) al que todos los hilos necesitan acceder de forma simultánea.
9.	Si empleamos un único cerrojo para proteger las secciones críticas de cierto objeto, se permitirá como máximo un hilo ejecutando dichas secciones críticas.
10.	En una sección crítica protegida por un `lock`, el protocolo de salida libera el `lock`, permitiendo que todos los hilos bloqueados en la entrada accedan al objeto.

ACTIVIDAD 1. OBJETIVO: Identificar los objetos compartidos y los tipos de sincronización que se requieren. Comprender por qué es necesaria la sincronización de las actividades. Identificar las partes de código que ocasionan condiciones de carrera.

PROBLEMA DEL PRODUCTOR-CONSUMIDOR: Considérese el caso de dos tareas que denominamos Productor y Consumidor, donde continuamente la primera tarea genera un dato, y se lo transfiere a la segunda que debe recibirlo y consumirlo. Se desea que ambas tareas se puedan llevar a cabo de forma concurrente, para lo cual se dispone de un buffer, donde el productor lo deja cuando lo genera, y de donde el consumidor lo recoge cuando esta dispuesto a aceptar un nuevo dato. Considérese que en esta versión simplificada, **el buffer está representado como una caja con cabida para solo un dato**. El Consumidor no debe recoger el dato antes de que el Productor lo haya generado.

En la página siguiente está el código asociado a este problema (también está accesible en el PoliformaT de la asignatura).

1) Identifique el/los objetos compartidos, los hilos y tipos de sincronización requerida, explicando brevemente de qué forma llevar a cabo la sincronización.

Objeto(s) Compartido(s)	
Hilos	
Tipos de sincronización	

2) En el código *Java* proporcionado (incluido también en la siguiente página) se producen algunos problemas relacionados con el uso de memoria compartida. ¿Qué problemas son? ¿Cómo podemos observarlos?

3) ¿Se pueden producir condiciones de carrera? Proporcione un ejemplo de traza correcta, es decir, sin condiciones de carrera. Y proporcione también un ejemplo de traza incorrecta.

4) Actualice el código para dar solución a la **exclusión mutua**.

Problema Productor – Consumidor (Ojo!! Sincronización incorrecta)

```
public class ProducerConsumer
{
     public static void main(String[] args)
     {
          Box c = new Box();
          Consumer c1 = new Consumer(c, 1);
          Producer p1 = new Producer(c, 1);

          c1.start();
          p1.start();

          try{
             c1.join();
             p1.join();
          }catch(InterruptedException e) { }
     }
}
```

```
public class Box
{
   private int content =0;
   private boolean full = false;

   public int get()
   {
          int value = content;
          content = 0;
          full = false;
          return value;
   }
   public void put(int value)
   {
          full = true;
          content = value;
   }
}
```

```
public class Consumer extends Thread
{
     private Box box;
     private int cname;
     public Consumer(Box c, int name)
     {
          box = c;
          cname = name;
     }

      public void run()
     {  for (int i=1; i<=10; i++){
          int value = 0;
          value = box.get();
          System.out.println("Consumer #" +
             cname + " gets: " + value);
          try {
            Thread.sleep((int)(Math.random() *
              100));
          } catch (InterruptedException e) { }
        }
     }
}
```

```
public class Producer extends Thread
{
     private Box box;
     private int prodname;
     public Producer (Box c, int name)
     {
          box = c;
          prodname = name;
     }

      public void run()
     {  for (int i=1; i<=10; i++){
          box.put(i);
           System.out.println("Producer #" +
            prodname + " puts: " + i);
          try {
            Thread.sleep((int)(Math.random() *
              100));
          } catch (InterruptedException e) { }
        }
     }
}
```

ACTIVIDAD 2. OBJETIVOS: Identificar las partes de código que pueden ocasionar problemas y cómo protegerlas.

Se ha definido la clase *Counter* que permite realizar incrementos y decrementos de un contador. Además, la clase RaceCondition contiene el método main() que lanza cuatro hilos (2 incrementadores y 2 decrementadores) que actúan sobre la misma variable compartida (i.e. mismo Counter). **El hilo principal debe imprimir el valor final del contador tras la ejecución de los otros cuatro hilos.**

Aquí tenemos el código de estas cuatro clases: *Counter*, *Incrementer*, *Decrementer* and *RaceCondition*. En el *PoliformaT* de la asignatura tiene disponible el código Java asociado a este problema.

```
class Counter {
    private int c = 0;

    public void increment() {
        c++;
    }

    public void decrement() {
        c--;
    }

    public int value() {
        return c;
    }
}
```

```
public class RaceCondition
{
     public static void main(String[] args)
     {
       Counter c = new Counter();
       int loops=1000;
       System.out.println("Loops "+ loops );

       Incrementer inc1 = new Incrementer(c, 1, loops);
       Incrementer inc2 = new Incrementer(c, 2, loops);

       Decrementer dec1 = new Decrementer(c, 1, loops);
       Decrementer dec2 = new Decrementer(c, 2, loops);

       inc1.start(); inc2.start();
       dec1.start(); dec2.start();

       System.out.println("Final result: "+c.value() );
     }
}
```

```
public class Incrementer extends Thread
{  private Counter c;
   private int myname;
   private int cycles;
   public Incrementer(Counter count,
         int name, int quantity)
   {   c = count;
       myname = name;
       cycles=quantity;
   }

   public void run()
   { for (int i = 0; i < cycles; i++)
     { c.increment();
       try
       { sleep((int)(Math.random() * 100));
       } catch (InterruptedException e){ }
     }
    System.out.println("Incrementer #" + myname+"
has done "+cycles+" increments.");
   }
}
```

```
public class Decrementer extends Thread
{  private Counter c;
   private int myname;
   private int cycles;
   public Decrementer(Counter count,
        int name, int quantity)
   {   c = count;
       myname = name;
       cycles=quantity;
   }

   public void run()
   { for (int i = 0; i < cycles; i++)
     {  c.decrement();
        try
        { sleep((int)(Math.random() * 100));
        } catch (InterruptedException e) { }
     }
     System.out.println("Decrementer #" + myname + "
has done "+cycles+" decrements.");
   }
}
```

1) Analice el código anterior e indique qué problemas pueden aparecer al ejecutarlo. ¿Se pueden producir condiciones de carrera?

2) Explique cómo solucionar estos problemas utilizando sincronización.

3) Actualice el código Java para aplicar la sincronización y compruebe si se resuelven los problemas identificados.

ACTIVIDAD 3. OBJETIVO: Comprender la diferencia entre métodos sincronizados (*synchronized methods*) y sentencias sincronizadas (*synchronized statements*) que ofrece Java.

Dado el siguiente código Java, indique cómo proteger sus secciones críticas para permitir que el método *inc1()* se ejecute en **exclusión mutua**; que el método *inc2()* se ejecute también en **exclusión mutua**; pero que **ambos métodos** puedan ser **ejecutados de forma concurrente** (a la vez) por diferentes hilos, sin bloquearse.

```
public class MsLunch {
  private long c1, c2 = 0;
  public  void inc1() {
    c1++;
    System.out.println(Thread.currentThread().getName()+": c1 value= "+c1);
  }

  public  void inc2() {
     c2++;
     System.out.println(Thread.currentThread().getName()+": c2 value="+c2);
   }
}

public class ThreadA extends Thread
{   private MsLunch c;
    private String threadname;

    public ThreadA(MsLunch counter, String name)
    {    c=counter;
         threadname = name;
         this.setName(threadname);
    }
    public void run()
    {  System.out.println("Thread #" + threadname + " wants to use inc1");   c.inc1();
       Thread.yield();
       System.out.println("Thread #" + threadname + " wants to use inc2");   c.inc2();
       System.out.println("Thread #" + threadname + " finishes");
    }
 }

public class ThreadB extends Thread
{   private MsLunch c;
    private String threadname;

    public ThreadB(MsLunch counter, String name)
    {    c=counter;
         threadname = name;
         this.setName(threadname);
    }
    public void run()
    { System.out.println("Thread #" + threadname + " wants to use inc2");    c.inc2();
      Thread.yield();
      System.out.println("Thread #" + threadname + " wants to use inc1");    c.inc1();
      System.out.println("Thread #" + threadname + " finishes");
    }
}

public class NOTSynchronizedStatements
{   public static void main(String[] args)
     {   MsLunch c = new MsLunch();
         ThreadA tA = new ThreadA(c, "A");
         Thread tB= new ThreadB(c, "B");
         tA.start();
         tB.start();
     }
}
```

ACTIVIDAD 4. OBJETIVOS: Identificar las partes de código que pueden ocasionar problemas y cómo protegerlas. Identificar los tipos de sincronización que se requieren.

Observe las siguientes soluciones al ***problema del productor-Consumidor*** con buffer sencillo (Soluciones 1 a 4) . En todas las soluciones, sólo cambia la clase **Box**, el resto de clases son iguales a las de la actividad 1. Para cada solución, indique si resuelve todos los tipos de sincronización requeridos (i.e. **exclusión mutua** y **sincronización condicional**). En su caso, indique qué problemas se observan.

```
// SOLUCIÓN 1
public class Box
{
   private int content =0;
   private boolean full = false;

   public int get()
   {
         if (!full) Thread.yield();

         int value = content;
         content = 0;
         full = false;
         return value;
   }

   public void put(int value)
   {
         if (full) Thread.yield();
         full = true;
         content = value;
   }
}
```

```
// SOLUCIÓN 2
public class Box
{
   private int content =0;
   private boolean full = false;

   public int get()
   {
         while (!full) Thread.yield();

         int value = content;
         content = 0;
         full = false;
         return value;
   }

   public void put(int value)
   {
         while (full) Thread.yield();
         full = true;
         content = value;
   }
}
```

```
// SOLUCIÓN 3
public class Box
{
   private int content =0;
   private boolean full = false;

   public synchronized int get()
   {
         int value = content;
         content = 0;
         full = false;
         return value;
   }

   public synchronized void put(int value)
   {
         full = true;
         content = value;
   }
}
```

```
// SOLUCIÓN 4
public class Box
{
   private int content =0;
   private boolean full = false;

   public synchronized int get()
   {
        while (!full) Thread.yield();
        int value = content;
         content = 0;
         full = false;
         return value;
   }

   public synchronized void put(int value)
   {
          while (full) Thread.yield();
          full = true;
          content = value;
   }
}
```