Actividades UD2.-Cooperación entre hilos

Concurrencia y Sistemas Distribuidos

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~~. **paso de mensajes**

   Otra opción: dos hilos del mismo proceso

2. Se produce una condición de carrera cuando se realiza una modificación de la memoria compartida que da lugar a un estado ~~consistente~~. **texto**

3. En el problema Lectores-Escritores se requiere acceso en exclusión mutua ~~solamente~~ para los escritores entre sí. **y para escritor con lector**

4. La sincronización condicional implica suspender a un hilo ~~durante un tiempo máximo especificado en la condición~~ hasta que se cumpla una determinada condición

5. La sección crítica es un fragmento de código que puede provocar condiciones de carrera.

   CIERTO

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~~. **progreso**

   hay que emplear un protocolo de entrada y un protocolo de salida que juntos garanticen… [?]

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~~. **lo cierra**

8. El problema de la sección crítica aparece cuando varios hilos comparten un objeto ~~in~~mutable (~~constante~~) al que todos los hilos necesitan acceder de forma simultánea. **objeto que puede ser modificado**

9. Si empleamos un único cerrojo para proteger las secciones críticas de cierto objeto, se permitirá como máximo un hilo ejecutando dichas secciones críticas.

   CIERTO

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.

    Uno de los hilos bloqueados

Cuestión 1 (afirmaciones 1-5)

- Indique si las siguientes afirmaciones son V/F, modifique lo necesario para que sean 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.

Cuestión 1 (afirmaciones 1-5)

- Indique si las siguientes afirmaciones son V/F, modifique lo necesario para que sean ciertas:

F	1. Dos hilos de ~~dos procesos distintos~~ pueden comunicarse entre sí utilizando memoria compartida. => dos hilos del mismo proceso
F	2. Se produce una condición de carrera cuando se realiza una modificación de la memoria compartida que da lugar a un estado ~~consistente.~~ => inconsistente
F	3. En el problema Lectores-Escritores se requiere acceso en exclusión mutua ~~solamente~~ para los escritores entre sí. => también para escritor – lector entre sí
F	4. La sincronización condicional implica suspender a un hilo ~~durante un tiempo máximo especificado en la condición~~. => hasta que otro hilo lo reactive porque ya se cumple la condición
V	5. La sección crítica es un fragmento de código que puede provocar condiciones de carrera.

Cuestión 1 (afirmaciones 6-10)

- Indique si las siguientes afirmaciones son V/F, modifique lo necesario para que sean ciertas:

	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.

Cuestión 1 (afirmaciones 6-10)

- Indique si las siguientes afirmaciones son V/F, modifique lo necesario para que sean ciertas:

F	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~~. => los protocolos de **entrada** y **salida** deben garantizar juntos las propiedades de exclusión mutua, espera limitada y **progreso**
F	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~~. => lo cierra
F	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.
V	9. Si empleamos un único cerrojo para proteger las secciones críticas de cierto objeto, se permitirá como máximo un hilo ejecutando dichas secciones críticas.
F	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. => uno de los hilos

Actividad 1

- 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.
- 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	

Actividad 1

- 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.
- 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)	El buffer y el estado del buffer. Objeto Box *c* (atributos *content* y *full*)
Hilos	Productores y Consumidores
Tipos de sincronización	Exclusión mutua en el acceso a los objetos compartidos. Sincronización condicional: el productor debe esperar a poner un nuevo dato si el buffer está lleno, y el consumidor debe esperar a recoger un dato si el buffer está vacío.

Actividad 1

2) En el código, ¿qué problemas relacionados con el uso de memoria compartida se producen?. ¿Cómo podemos observarlos?

```
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;
    }
}
```

Ambos hilos (c1 y p1) acceden concurrentemente al mismo objeto compartido (c) sin utilizar ningún mecanismo de sincronización. Pueden producirse errores de interferencias entre hilos y de inconsistencia de memoria

```
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) { }
      }
    }
}
```

Pueden observarse:

- si se obtiene un 0, se está accediendo a la caja vacía;
- Se puede no obtener todo lo que se ha puesto en la caja
- Se puede no poner algún elemento

Actividad 1

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.

```
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;
   }
}
```

Sí se pueden producir condiciones de carrera.
Traza correcta aquella en la que se obtiene todo lo que se pone (del 1 al 10).

```
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) { }
       }
    }
}
```

Traza correcta

```
Producer #1 puts: 1
Consumer #1 gets: 1
Producer #1 puts: 2
Consumer #1 gets: 2
Producer #1 puts: 3
Consumer #1 gets: 3
Producer #1 puts: 4
Consumer #1 gets: 4
Consumer #1 gets: 5
Producer #1 puts: 5
Producer #1 puts: 6
Consumer #1 gets: 6
Producer #1 puts: 7
Consumer #1 gets: 7
Producer #1 puts: 8
Consumer #1 gets: 8
Producer #1 puts: 9
Consumer #1 gets: 9
Producer #1 puts: 10
Consumer #1 gets: 10
```

Actividad 1

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.

```
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;
   }
}
```

Sí se pueden producir condiciones de carrera.
Traza correcta aquella en la que se obtiene todo lo que se pone (del 1 al 10).

```
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) { }
       }
    }
}
```

Traza incorrecta

```
Consumer #1 gets: 0
Producer #1 puts: 1
Consumer #1 gets: 1
Producer #1 puts: 2
Producer #1 puts: 3
Consumer #1 gets: 3
Producer #1 puts: 4
Consumer #1 gets: 4
Consumer #1 gets: 0
Consumer #1 gets: 0
Producer #1 puts: 5
Consumer #1 gets: 5
Producer #1 puts: 6
Consumer #1 gets: 6
Producer #1 puts: 7
Producer #1 puts: 8
Consumer #1 gets: 8
Producer #1 puts: 9
Producer #1 puts: 10
Consumer #1 gets: 10
```

Actividad 1

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

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

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

Actividad 2

Se producen Condiciones de carrera: los hilos acceden a estados intermedios del contador.

```
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.");
   }
}
```

Condición de carrera: `main` obtiene un valor de **c** anterior a que se haya actualizado el contador por parte de todos los hilos

Actividad 2

2) Explique cómo solucionar estos problemas utilizando sincronización.

Proporcionando **exclusión mútua** en el acceso al contador

```
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.");
   }
}
```

`main` debe esperar la finalización de todos los hilos antes de escribir el valor.

3) Actualice el código Java para aplicar la sincronización.

exclusión mútua → **métodos synchronized**

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

public synchronized void increment() {
        c++;
    }

public synchronized void decrement() {
        c--;
    }

public synchronized 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();
   try {inc1.join(); inc2.join(); dec1.join();
   dec2.join()} catch (InterruptedException e) {};
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.");
   }
}
```

```
{  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.");
   }
}
```

Thread.join garantiza la relación **ocurre-antes**.
Los efectos del código del hilo que termina son visibles por el hilo que realizó el join.
Ponerlo en un bloque try … catch …

Actividad 3

- 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 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);
  }
}
```

Actividad 3

```
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();
     }
}
```

```
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");
    }
}
```

Actividad 3 → Solución

- 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 a la vez por diferentes hilos, sin bloquearse.

```
public class MsLunch {
  private long c1, c2 = 0;
  private Object l1=new Object();
  private Object l2=new Object();
  public  void inc1() {
   synchronized (l1) {
     c1++;
    System.out.println(Thread.currentThread().getName()+": c1 value= "+c1);
   }
  }
  public  void inc2() {
   synchronized (l2) {
     c2++;
     System.out.println(Thread.currentThread().getName()+": c2 value="+c2);
   }
  }
}
```

Actividad 4

- Observe las siguientes soluciones al problema del productor-Consumidor con buffer sencillo. 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.

```
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 4

```
// 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;
    }
}
```

- No soluciona la exclusión mutua.
- Proporciona una solución parcial a la sincronización condicional, con el if (!full) del get, o el if(full) del put, pero esta solución no funciona bien ya que cuando el thread vuelve a ejecución continúa después del if, por lo que no vuelve a comprobar si la condición es correcta o no.
- Traza a analizar:
  - entra cons, cede CPU
  - entra prod, pone 1, sleep largo,
  - entra cons, obtiene 1, sleep corto, entra cons, cede CPU pero como es el único preparado continúa => **obtiene 0**, sleep largo
  - entra prod, pone 2, sleep corto, entra prod, cede CPU, pero es el único preparado, continúa y **pone 3**, machacando el valor 2 que nunca podrá obtener el cons.

Actividad 4

```
// 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;
   }
}
```

- Corrige el error anterior, aunque continúe por ser el único preparado, comprueba la condición de lleno/vacío otra vez.
- No soluciona la exclusión mutua.

- Trazas a analizar:
  - entra prod, le interrumpen en (*) antes de la asignación content=value por lo que el contenido es 0 pero full está true
  - entra cons: como está llena obtiene un 0

  - entra prod, acaba la asignación y pone el 1.
  - entra prod, como está vacío pone el 2, sin que el consumidor haya obtenido el 1.

Actividad 4

```
// 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;
   }
}
```

- Se soluciona la exclusión mutua (con los dos métodos etiquetados con "synchronized").
- Pero no aporta solución a las condiciones de sincronización.
- Se cumplen las condiciones de progreso y equidad, al no quedarse ningún hilo bloqueado.
- Pero se pueden producir condiciones de carrera, al no quedarse los productores a la espera cuando se llena el buffer, ni los consumidores cuando el buffer está vacío.

Actividad 4

```
// 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;
   }
}
```

- Soluciona la exclusión mutua y
- la sincronización condicional, pero esta solución no es correcta.
- **No** se cumple la condición de **progreso**, ya que los hilos se quedan bloqueados entre sí.

- Traza a analizar con caja vacía:
  - entra cons, coge la exc. mutua de su método get (cierra el lock implícito) y como está vacía cede el procesador (**sin liberar el lock implícito**) =>
  - el productor no podrá entrar en su método "put" y escribir en el buffer. Ambos hilos se quedan bloqueados.