GESTIÓN DE HILOS EN JAVA

Creación de hilos

Hay dos opciones:

- La clase Thread

```
class NombreHilo extends Thread {
        // propiedades, constructores y métodos de la clase
        public void run() {
                // acciones que lleva a cabo el hilo
        }
}
```

- La interfaz Runnable

```
Class NombreHilo implements Runnable {
        // propiedades, constructores y métodos de la clase
        public void run() {
                // acciones que lleva a cabo el hilo
        }
}
```

Ejemplos ()

Ejemplo 1 (UD2Ej01): Identificación del hilo de ejecución

Todo proceso en ejecución tiene al menos un hilo aunque no se cree explícitamente

```
   /*Este programa Identifica el hilo que ejecuta el método main() de la
     típica aplicación de consola "¡Hola mundo!"
    * Se utilizan para ello los métodos: currentThread() y getName()
    * de la clase Thread
    */

public class UD2Ej01 {
    public static void main(String[] args) {

    System.out.println("¡Hola mundo!\n");
    //imprime "¡Hola mundo!" en la Salida
    Thread miHilo = Thread.currentThread();
    //obtiene el hilo donde se está ejecutando este método mediante la
    //función Thread.currentThread(), y lo almacena en la variable
    //local miHilo

    //imprime el nombre del hilo en la Salida (función getName())
    System.out.println("Por defecto, el hilo que ejecuta el método main() "
            +"de mi programa se llama '" + miHilo.getName() + "'\n");

  }
}
```

Ejemplo 2 (UD2Ej02): Creación de un hilo por extensión de la clase Thread

Se crea una clase hilo nueva por extensión de Thread. La nueva clase ha de sobrecargar el método run con el código a ejecutar por el hilo y que será ejecutado cuando se invoque al método start del hilo creado

```
public class  UD2Ej02 extends Thread {
 //clase  que extiende a Thread
    public void run() {
    // se redefine el método run() con el código asociado al hilo
        System.out.println("¡Saludo desde un hilo extendiendo thread!");
    }
    public static void main(String args[]) {
        UD2Ej02 hilo1=new UD2Ej02();
        //se crea un objeto Thread, el hilo hilo1
        hilo1.start();
        //invoca a start() y pone en marcha el hilo hilo1
    }
  }
```

Ejemplo 3 (UD2Ej03): Creación de un hilo implementando la interfaz Runnable

Se añade la funcionalidad de hilo a una clase implementando la interfaz runnable y creando un hilo con un objeto de dicha clase como parámetro al constructor

La funcionalidad del hilo se codificará también en su método run y será ejecutado al invocar el método start del hilo que se crea

```
public class UD2Ej03 implements Runnable {
      //clase que implementa a Runnable
    public void run() {
    //se redefine el método run()con el código asociado al hilo
        System.out.println("¡Saludo desde un hilo creado con Runnable!");
    }
    public static void main(String args[]) {
      UD2Ej03  miRunnable=new UD2Ej03();
       //se crea un objeto  Saludo
       Thread hilo1= new Thread(miRunnable);
       //se crea un objeto Thread (el hilo hilo1) pasando como argumento
      // al constructor un objeto Saludo
        hilo1.start();
       //se invoca al método start() del hilo hilo1
    }
}
```

Ejemplo 4 (UD2Ej04): Aplicación con varios hilos Thread y Runnable separando las clases de los hilos de la clase que los ejecuta

Aquí tenemos el código de tres clases, una para crear un hilo con interfaz runnable, otra para crear un hilo extensión de thread y otra main para hacer uso de las anteriores con la función main

Obsérvese que los hijos no se ejecutan en el orden en que se crean ya que en lenguaje Java no es determinístico en la ejecución de los hilos

Clase Hilo_Runnable:

```
public class Hilo Runnable implements Runnable {
    //clase que implementa Runnable
    public void run() {
        //redefinimos run() con el código asociado al hilo
        for (int i = 1; i <= 5; i++) {
            System.out.println("  Hilo_Runnable");
        }
    }
}

/**************************************/
```

Clase Hilo_Thread:

```
public class Hilo_Thread extends Thread {
//clase que extiende a Thread con 2 constructores

    String nombre = "Hilo derivaThread";

    public Hilo_Thread(String nb) {
        //constructor 1
        nombre = nb;
    }

    public Hilo_Thread() {
        //constructor 2
    }

    @Override
    public void run() {
        //redefinimos run() con el código asociado al hilo
        for (int i = 1; i <= 5; i++) {
            System.out.println(nombre);
        }
    }
}

/*********************************************/
```

Clase Main:

```
public class Main {

   public static void main(String[] args) {
        //creamos 2 hilos del tipo Hilo_Thread con 2 constructores
        //diferentes
        Thread hilo1 = new Hilo_Thread("Isabel");
        Thread hilo2 = new Hilo_Thread();

        //creamos un hilo Runnable en un paso
        Thread hilo3 = new Thread(new Hilo Runnable());

        //ponemos en marcha los 3 hilos
        hilo1.start();
        hilo2.start();
        hilo3.start();
    }
}
```

Ejemplo 5 (UD2Ej05): Estados de un hilo

En este ejemplo podemos ver el estado en que se encuentra un hilo en distintos momentos de su vida haciendo uso del método getState(). Asímismo determinamos si está vivo mediente el método isAlive()

Clase Hilo_Auxiliar:

```
public class Hilo_Auxiliar extends Thread{
//código del hilo
  @Override
  public void run(){
    for(int i=10;i>=1;i--)
      System.out.print(i+",");
  }
}
```

Clase Main:

```
import java.util.logging.Level;
import java.util.logging.Logger;

/**************************************************************************/

public class Main {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        Hilo Auxiliar hilo1 = new Hilo Auxiliar();
        //Crea un nuevo hilo. El hilo está en estado Nuevo (new)

        System.out.println("Hilo Auxiliar Nuevo: Estado=" + ilo1.getState()
                + ",¿Vivo? isAlive()=" + hilo1.isAlive());
        //Obtenemos el estado del thread hilo1 y si está vivo o no

        hilo1.start();
        //Inicia el thread hilo1 y pasa al estado Ejecutable

        System.out.println("Hilo Auxiliar Iniciado: Estado="
+ hilo1.getState() + ",¿Vivo? isAlive()=" + hilo1.isAlive() + "\n");

        try {
            hilo1.join();
            //espera a que el thread hilo1 muera
        } catch (InterruptedException e) {
            System.out.println(e);
        }
        System.out.println("\n Hilo Auxiliar Muerto: Estado="
                + hilo1.getState()
                + ",¿Vivo? isAlive()=" + hilo1.isAlive());
    }
}
```

Ejemplo 6 (UD2Ej06): Gestión de prioridades de hilo

En este ejemplo se usan los métodos getPriority() y setPriority() para obtener y modificar, respectivamente, la prioridad de un hilo

Además se usa el método yield() para ceder voluntariamente el control de forma que el planificador pueda ejecutar otro hilo

Clase Hilo:

```
public class Hilo extends Thread {

/*************************************************************************
     * constructor por defecto
     */
    public Hilo() {
        //hereda la prioridad del hilo padre
    }
/*************************************************************************
     * constructor personalizado
     */
    public Hilo(int prioridad) {

        //establece la prioridad indicada
        this.setPriority(prioridad);
    }
/*************************************************************************
     * ejecuta una tarea pesada
     */
    @Override
    public void run() {

        //cadena
        String strCadena = "";

        //agrega 20000 caracteres a una cadena vacía
        for (int i = 0; i < 20000; ++i) {
            //imprime el valor en la Salida
            strCadena += "A";
            yield();
            //yield()sugiere al planficador Java que puede seleccionar otro
hilo,

        }

        System.out.println("Hilo de prioridad " + this.getPriority()
                + " termina ahora");
    }
}
```

Clase Main:

```
public class Main {

/**************************************************************************
     *
     *
     * @param args the command line arguments
     */
    public static void main(String[] args) {

        int contador = 5;

        //vectores para hilos de distintas prioridades
        Thread[] hiloMIN = new Thread[contador];
        Thread[] hiloNORM = new Thread[contador];
        Thread[] hiloMAX = new Thread[contador];

        //crea los hilos de prioridad mínima
        for (int i = 0; i < contador; i++) {
            hiloMIN[i] = new Hilo(Thread.MIN_PRIORITY);
        }

        //crea los hilos de prioridad normal
        for (int i = 0; i < contador; i++) {
            hiloNORM[i] = new Hilo();
        }

        //crea los hilos de máxima prioridad
        for (int i = 0; i < contador; i++) {
            hiloMAX[i] = new Hilo(Thread.MAX_PRIORITY);
        }

        System.out.println("Hilos en proceso, espera......\nLos de mayor "
                + "prioridad tienden a terminar antes...\n");

        //inicia los hilos
        for (int i = 0; i < contador; i++) {
            hiloMIN[i].start();
            hiloNORM[i].start();
            hiloMAX[i].start();
        }

    }
```

Ejemplo 7 (UD2Ej07): Orden de ejecución de hilos

En este ejemplo se puede comprobar si el SO evita los procesos egoístas o no. Si no lo hace se ejecutará el primer hilo, Rojo, y escribirá sus 100 veces sin dejar al segundo hilo comenzar. En caso contrario se irán intercalando ejecuciones de ambos hilos

Con la variante de yield() conseguimos que en cada iteración el proceso ceda "voluntariamente" la ejecución a otros

```
public class Color extends Thread {
   //clase que extiende a Thread
    String color;
    public Color (String c){
        color=c;
    }
   public void run(){
       //se imprime 100 veces el valor de: color + i
       for(int i=1;i<=100;i++)
            System.out.println(color + i);

      /* Variante con cesión del turno voluntariamente
       for(int i=1;i<=100;i++) {
            System.out.println(color + i);
            yield();} //llamada a yield()
      */
    }
}

public class Main {
 public static void main(String[] args) {
       //se crean dos hilos: hrojo y hazul
       Color hrojo = new Color ("Rojo");
       Color hazul = new Color ("Azul");
       //se inician los hilos para su ejecución
       hrojo.start();
       hazul.start();
    }
}
```

Ejemplo 8 (UD2Ej08): Jardines. Secciones críticas y condiciones de carrera

En este ejemplo se implementa el supuesto de un jardín que tiene una puerta de entrada y otra de salida y se trata de contabilizar las personas que hay en el jardín en cada momento

Unos hilos simularán la Entrada y otros la Salida al jardín, y todos comparten un recurso, la variable cuenta, que se incrementa o decrementa en uno según sea Entrada o Salida

Se supondrá que inicialmente ya hay 100 personas en el jardín

Clase RecursoJardin:

```
public class RecursoJardin {
    //clase que simula las entradas y las salidas al Jardín
    private int cuenta; //para contar las entradas y salidas al Jardín
    public RecursoJardin() {
        cuenta = 100; //inicalmente hay 50 personas en le jardín
    }
    public void incrementaCuenta() {
        //método que increamenta en 1 la varibale cuenta
        System.out.println("hilo " + Thread.currentThread().getName()
                + "----- Entra en Jardín");
        //muestra el hilo que entra en el método
        cuenta++;
        System.out.println(cuenta + " en jardín");
        //cuenta cada acceso al jardín y muestra el número de accesos
    }
   public  void  decrementaCuenta() {
        //método que decrementa en 1 la varibale cuenta
        System.out.println("hilo " + Thread.currentThread().getName()
                + "----- Sale de Jardín");
        //muestra el hilo que sale en el método
        cuenta--;
        System.out.println(cuenta + " en jardín");
        //cuenta cada acceso al jardín y muestra el número de accesos
    }
}
```

Clase Entra_Jardin:

```
public class Entra_Jardin extends Thread {
//clase derivada de Thread que define un hilo
    private RecursoJardin jardin;

    public Entra_Jardin(String nombre, RecursoJardin j) {
        this.setName(nombre);
        this.jardin = j;
    }

    @Override
    public void run() {
        jardin.incrementaCuenta();
        //invoca al método que incrementa la cuenta de accesos al jardín
    }
}
```

Clase Sale_Jardin:

```
public class Sale_Jardin extends Thread {
//clase derivada de Thread que define un hilo
    private RecursoJardin jardin;
    public Sale Jardin(String nombre, RecursoJardin j) {
        this.setName(nombre);
        this.jardin = j;
    }

    @Override
    public void run() {
        jardin.decrementaCuenta();
        //invoca al método que decrementa la cuenta de accesos al jardín
    }
}
```

Clase Main:

```
public class Main {
    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        RecursoJardin jardin = new RecursoJardin();
        //crea un objeto RecursoJardín

        for (int i = 1; i <= 10; i++) {
            (new Entra Jardin("Entra" + i, jardin)).start();
        }//entrada de 10 hilos al jardín

        for (int i = 1; i <= 15; i++) {
            (new Sale Jardin("Sale" + i, jardin)).start();
        }//salida de 15 hilos al jardín
    }
}
```

Al ejecutar este ejemplo veremos que no se obtienen resultados correctos ya que no se ha realizado sincronización alguna entre los hilos

Ejemplo 9 (UD2Ej09): Jardines. Secciones críticas y condiciones de carrera. Sincronización con monitores a nivel de método

Se soluciona el problema anterior usando monitores para conseguir acceso exclusivo a los métodos que modifican el valor del contador del jardín

```
public class RecursoJardin {
//clase que simula las entradas y las salidas al Jardín
    private int cuenta; //para contar las entradas y salidas al Jardín
    public RecursoJardin() {
        cuenta = 100; //inicalmente hay 100 personas en le jardín
    }
    public synchronized void incrementaCuenta() {
        //método que increamenta en 1 la varibale cuenta
        System.out.println("hilo " + Thread.currentThread().getName()
                + "----- Entra en Jardín");
        //muestra el hilo que entra en el método
        cuenta++;
        System.out.println(cuenta + " en jardín");
        //cuenta cada acceso al jardín y muestra el número de accesos
    }
   public synchronized void  decrementaCuenta() {
        //método que decrementa en 1 la varibale cuenta
        System.out.println("hilo " + Thread.currentThread().getName()
                + "----- Sale de Jardín");
        //muestra el hilo que sale en el método
        cuenta--;
        System.out.println(cuenta + " en jardín");
        //cuenta cada acceso al jardín y muestra el número de accesos
    }
}
```

Ejemplo 10 (UD2Ej10): Jardines. Secciones críticas y condiciones de carrera. Sincronización con monitores a nivel de bloque de código

Variante al ejemplo anterior en el que los monitores no se aplican a métodos completos sino a bloques de código, situación que puede ser necesaria si por ejemplo no podemos editar el método en cuestión

Partiendo del Ejemplo 8 sólo modificamos las clases Sale_Jardin y Entra_Jardin como sigue:

```
public class Sale_Jardin extends Thread {
//clase derivada de Thread que define un hilo
    private RecursoJardin jardin;
    public Sale_Jardin(String nombre, RecursoJardin j) {
        this.setName(nombre);
        this.jardin = j;
    }

    @Override
    public void run() {
      synchronized(jardin) {
            jardin.decrementaCuenta();
      }
        //invoca al método que decrementa la cuenta de accesos al jardín
    }
}

public class Entra_Jardin extends Thread {
//clase derivada de Thread que define un hilo
    private RecursoJardin jardin;

    public Entra_Jardin(String nombre, RecursoJardin j) {
        this.setName(nombre);
        this.jardin = j;
    }

    @Override
    public void run() {
      synchronized(jardin){
            jardin.incrementaCuenta();
      }
        //invoca al método que incrementa la cuenta de accesos al jardín
    }
}
```

Ejemplo 11 (UD2Ej11): Lectores-Escritores. Comunicación entre hilos usando wait() y notify()

A una base de datos pueden acceder concurrentemente varios hilos lectores de un registro y varios hilos escritores que insertan o modifican un registro de una tabla.

Cuando un escritor accede a la base de datos, ésta debe estar libre, es decir no debe haber ningún lector o escritor utilizándola, ya que el escritor modifica los datos.

Si llega un lector y hay algún escritor deberá esperar a que el escritor termine.

Si llega un escritor debe esperar a que terminen todos los lectores y escritores que haya para poder entrar.

Clase Semaforo: Código para controlar el acceso de los lectores y escritores y su coordinación

```
/**************************************************************************
****
 * clase que proporciona el objeto Semaforo encargado de controlar el
acceso de lectores y escritores a los datos también proporciona los métodos
para notificar al semáforo notificar que la finalización de un lector y un
escritor y así poder acualizar el estado del semáforo
 */
public class Semaforo {

    public final static int LIBRE = 0;
    //indica que no hay lectores leyendo, ni ningún escritor escribiendo.
    //En este estado pueden entrar lectores a leer, o un escritor a
escribir
    public final static int CON_LECTORES = 1;
    //constante que indica que hay lectores leyendo. Puede entrar un nuevo
    //lector a leer, pero no puede entrar ningún escritor a escribir
    public final static int CON_ESCRITOR = 2;
    //constante que indica que hay escritores escribiendo. En este estado,
no
    //puede entrar ningún lector a leer, ni nigún escritor a escribir
    private int estado = LIBRE;
    //estado del semáforo (inicialmente: libre)
    private int tLectores = 0;
    //número de lectores (inicialmente: ninguno)


/**************************************************************************
     * método que da acceso a la lectura de datos
     */
    public synchronized void accesoLeer() {
        //método sincronizado. Sólo un hilo lo usa a la vez
        String nombre = Thread.currentThread().getName();
        //guarda el nombre del hilo que se hace con el método
        if (estado == LIBRE) {
            //BD sin lectores ni escritores. Puede netrar a leer
            System.out.println("BD:" + estado + " " + tLectores + "L " +
nombre
                    + " entra a leer.");
            //mensaje para comprobar el funcionamiento
            estado = CON_LECTORES;
            //cambia estado, yahay lector
        } else if (estado != CON_LECTORES) {
```

```
            //si no está libre, ni con lectores
            while (estado == CON_ESCRITOR) {
                try {
                    System.out.println("BD:" + estado + " " + tLectores +
"L "
                            + nombre + " trata de leer.ESPERA");
                    //mensaje para comprobar el funcionamiento
                    wait();
                    //pone en espera al hilo que intenta leer datos
                } catch (InterruptedException e) {
                    System.out.println(e);
                }
            }
            System.out.println("BD:" + estado + " " + tLectores + "L "
                    + nombre + " entra a leer.");
            //mensaje para comprobar el funcionamiento
            estado = CON_LECTORES;
            //cambia estado, ya hay lector
        } else {//en este punto el estado es CON LECTORES
            System.out.println("BD:" + estado + " " + tLectores + "L "
                    + nombre + " entra a leer.");
            //mensaje para comprobar funcionamiento
        }
        tLectores++;
        //otro lector más
        System.out.println("BD:" + estado + " " + tLectores + "L "
                + nombre + " Leyendo.....");
        //mensaje para comprobar funcionamiento
    }


/**************************************************************************
     * método que da acceso para escribir datos si el estado de la BD lo
permite
     */
    public synchronized void accesoEscribir() {
        String nombre = Thread.currentThread().getName();
        //guarda el nombre del hilo que se hace con el método
        if (estado == LIBRE) {
            //sin lectores ni escritores
            System.out.println("BD:" + estado + " " + tLectores + "L "
                    + nombre + " entra a escribir.");
            //mensaje para comprobar el funcionamiento
            estado = CON_ESCRITOR;
            //cambia estado
        } else {//si no está libre
            while (estado != LIBRE) {
                //miestras BD está ocupada con lectores, o con un escritor
                try {
                    System.out.println("BD:" + estado + " " + tLectores +
"L "
                            + nombre + " trata de escribir.ESPERA");
                    //mensaje para comprobar funcionamiento
                    wait();
                    //pone en espera al hilo que intenta escribir datos
                } catch (InterruptedException e) {
                    System.out.println(e);
                }
            }// el estado ahora es LIBRE
            System.out.println("BD:" + estado + " " + tLectores + "L "
                    + nombre + " entra a escribir.");
```

```
            //mensaje para comprobar el funcionamiento
            estado = CON_ESCRITOR;
            //cambia estado
        }
        System.out.println("BD:" + estado + " " + tLectores + "L "
                + nombre + " Escribiendo..");
        //mensaje para comprobar el funcionamiento
    }


/**********************************************************************
     * método que invoca un HiloEscritor al terminar de escribir, para
     * actualizar el estado del semáforo y en su caso notificarlo a los
hilos
     * en espera.
     * Por supuesto, no se permite que dos hilos ejecuten estas
instrucciones
     * a la vez
     */
    public synchronized void escrituraFinalizada() {
        estado = LIBRE;
        //cambia estado
        System.out.println(Thread.currentThread().getName() + ": Ya ha
escrito");
        //mensaje para comprobar el funcionamiento
        notify();
        //notifica a los hilos en espera que ya ha finalizado
    }


/**********************************************************************
     * método que que invoca un HiloLector cuando termina de escribir, para
     * actualizar el estado del semáforo y en su caso notificarlo a los
hilos
     * en espera
     *
     * por supuesto, no se permite que dos hilos ejecuten estas
instrucciones
     * a la vez
     */
    public synchronized void lecturaFinalizada() {
        System.out.println(Thread.currentThread().getName() + ": Ya ha
leido");
        //mensaje para comprobar el funcionamiento
        tLectores--;
        //un lector menos leyendo
        if (tLectores == 0) {
            //no hay lectores en la BD
            estado = LIBRE;
            //cambia el estado
            notify();
            //notifica a los hilos en espera que ya ha finalizado
        }
    }
}
```

Clase HiloLector: Representa los hilos que pretenden leer en la BD

```
public class HiloLector extends Thread {

    private Semaforo semaforo;
    //semaforo de control de acceso recibido por el constructor. Le
proporciona
    //al hilo el método de lectura de los datos, así como el método para
    //actualizar el estado cuando finaliza esa lectura


/**************************************************************************
     * constructor: se le pasa el nombre y el semáforo de control */
    public HiloLector(String nombre, Semaforo s) {
        this.setName(nombre);
        this.semaforo = s;
    }

/**************************************************************************
     * el método run() del hilo que lee los datos */
    @Override
    public void run() {
        System.out.println(getName() + ": Intentando leer");
        //mensaje de salida para comprobar el funcionamiento
        semaforo.accesoLeer();
        //el hilo ha leido
        try {
            sleep((int) (Math.random()) * 50);
        } catch (InterruptedException e) {
            System.out.println(e);
        }
        //duerme al hilo antes de que éste comunique que ha finalizado,
para
        //poder ver accesos fallidos, con fines de comprobar funcionamiento
        semaforo.lecturaFinalizada();
        //comunica al semáforo la finalización de la lectura
    }
}
```

Clase HiloEscritor: Representa los hilos que pretenden escribir en la BD

```
public class HiloEscritor extends Thread {

    private Semaforo semaforo;
    //semaforo de control de acceso recibido por el constructor. Le
proporciona
    //al hilo el método de acceso para escribir datos, así como el método
para
    //actualizar su estado cuando finaliza esa escritura


/**************************************************************************
/
    //constructor: se le pasa el nombre y el semáforo
    public HiloEscritor(String nombre, Semaforo s) {
        this.setName(nombre);
        this.semaforo = s;
    }
    @Override
```

```
    public void run() {
    //método con el comportamiento del hilo
        System.out.println(getName() + ": Intentando escribir");
        //mensaje para la Salida y comprobar funcionamiento
        semaforo.accesoEscribir();
        //el hilo ha escrito
        try {
            sleep((int) (Math.random()) * 50);
            //duerme el hilo un tiempo aleatorio antes de comunicar el fin
de
            //la lectura, para dar ocasión de que los demás hilos hagan
            //intentos fallidos de lectura/escritura y comprobar
funcionamiento
        } catch (InterruptedException e) {
            System.out.println(e);
        }
        semaforo.escrituraFinalizada();
        //comunica al semáforo la finalización de la escritura
    }
}
```

Clase Main:

```
/**********************************************************************
****
 * pone 10 lectores a leer y 3 escritores a escribir sobre los mismo datos.
 * Utiliza un objeto de la clase Semáforo para que todas esas acciones se
 * realicen coherentemente
 *
 */
public class Main {

/**********************************************************************
     * @param args the command line arguments
     */
    public static void main(String args[]) {
        Semaforo smfro = new Semaforo();
        //semáforo de control

        //pone 5 lectores a leer y 2 escritores a escribir, controlados por
        //el mismo semáforo
        for (int i = 1; i <= 5; i++) {
            new HiloLector("Lector" + i, smfro).start();
        }

        for (int i = 1; i <= 2; i++) {
            new HiloEscritor("Escritor" + i, smfro).start();
        }
    }

}
```

Ejemplo 12 (UD2Ej12): Interbloqueo de hilos

En el siguiente ejemplo se da una situación de interbloqueo entre dos hilos que esperan por un recurso que está bloqueado por otro hilo

```
class FicheroA {

 /**********************************************************************
  *  métodoA: simula el acceso al ficheroA con exclusión mutua. Duerme al
hilo y  después invoca al método que le permitirá o intentará el acceso al
ficheroB
  * @param b
  */
  public synchronized void metodoA(FicheroB b) {

    //imprime en la salida el nombre del hilo que ejecuta el método
    String name = Thread.currentThread().getName();
    System.out.println("Hilo " + name + " entra en fichero A");

    try {
      Thread.sleep(1000);
    } catch (Exception e) {}

    System.out.println("Hilo " + name + " intentando acceder a fichero_B");
    b.metodoB(this);
  }

  public synchronized void syn2() {

    System.out.println("Dentro de A.syn2");
  }
}

class FicheroB  {

/**********************************************************************
   métodoB: simula el acceso al ficheroB con exclusión mutua. Duerme al
hilo y  después invoca al método que le permitirá o intentará el acceso al
ficheroA
   * @param a
   */

  public synchronized void metodoB(FicheroA a) {

    //imprime en la salida el nombre del hilo que ejecuta el método
    String name = Thread.currentThread().getName();
    System.out.println("Hilo " + name + " entra en fichero_B");

    try {
      //lo duerme durante 1s
      Thread.sleep(1000);
    } catch (Exception e) {}

    //imprime en la Salida el intento de llamada al método metodoA() del
objeto a
    System.out.println("Hilo " + name + " intentando entrar en fichero_A");
    a.metodoA(this);
  }
}
```

```
public class Hilo1 extends Thread {
 //Declara dos objetos del tipos de cada fichero
  FicheroA a;
  FicheroB b;


/**********************************************************************
***
  constructor
   *
   * @param a: ficheroA
   * @param b: ficheroB
   */
  Hilo1(FicheroA a, FicheroB b) {
    this.a = a;
    this.b = b;
  }
/**********************************************************************
*****
 * Tarea del hilo: acceder al ficheroA
 */
  @Override
  public void run() {
    a.metodoA(b);
    //el hilo accede al ficheroA
  }
}


public class Hilo2 extends Thread {

  //Declara dos objetos del tipos de cada fichero
  FicheroA a;
  FicheroB b;


/**********************************************************************
**
   * constructor
   *
   * @param a: ficheroA
   * @param b: ficheroB
   */
  Hilo2(FicheroA a, FicheroB b) {
    this.a = a;
    this.b = b;
  }


/**********************************************************************
****
   * Tarea del hilo: acceder al ficheroB
   */
  @Override
  public void run() {

    b.metodoB(a);
    //el hilo accede al ficheroB.
  }
}
```

```
/**********************************************************************
 *Programa ejemplo de interbloqueo o deadlock.
 * Supongamos dos hilos y que cada hilo necesita privilegios exclusivos de
 * escritura en dos archivos distintos. El hilo1 podría abrir el archivoA
de
 * forma exclusiva y el hilo2 hacer lo mismo con el archivoB.
 * Estando el hilo1 en el aechivoA necesita acceso exclusivo al archivoB y
 * estando el hilo2 en archivoB necesita acceso exclusivo al archivoA.
 * Ambos hilos se obstaculizarán entre sí y se bloquean indefinidamente. Se
abrá
 * producido un interbloqueo.
 *
 * @author IMCG
 */
class Main {

 //Se crean ficheros de ambos tipos, A y B
  static FicheroA a = new FicheroA();
   static FicheroB b = new FicheroB();


  public static void main(String args[]) {
    //crea e inicia los hilos que ejecutarán los métodos synchronized, y
que
    //provocarán el interbloqueo
    Hilo1 hilo1 = new Hilo1(a, b);
    Hilo2 hilo2 = new Hilo2(a, b);
    hilo1.start();
    hilo2.start();
  }
}
```

Ejemplo 13 (UD2Ej13): Servidor web. Protección de secciones críticas usando semáforos de la clase Semaphore

Se trata de contabilizar los accesos actuales a un servidor web. Simularemos este hecho mediante un recurso compartido, la variable cuenta, que se incrementa en cada acceso nuevo al servidor. Esto se representa en la clase ServidorWeb

Cada Terminal es un hilo representado por la clase Hilo_Terminal. Se suponen 4 terminales con 10 personas desde cada una

Supondremos que en el momento inicial hay 0 accesos

Clase ServidorWeb:

```
public class ServidorWeb {
//clase que simula los accesos a un servidor
    private int cuenta;
    public ServidorWeb() {
        cuenta = 0;
    }
    public void  incrementaCuenta() {
        //método sincronizado (monitor)
        System.out.println("hilo " + Thread.currentThread().getName()
                + "----- Entra en Servidor");
        //muestra el hilo que entra en el Servidor.Para probar
funcionamiento
        cuenta++;
        //se incrementa la cuenta de accesos
        System.out.println(cuenta + " accesos");
        //muestra el número de accesos. Para probar funcionamiento
    }
}
```

Clase Hilo_Terminal:

```
import java.util.concurrent.Semaphore;
public class Hilo_Terminal extends Thread {
//clase derivada de Thread que define un hilo

    private ServidorWeb servidor;
    private Semaphore semaforo;

    public Hilo_Terminal(ServidorWeb s, Semaphore se) {
        this.servidor = s;
        this.semaforo = se;
    }

    @Override
    public void run() {
        //la tarea del hilo es invocar a incrementaCuenta()simulando un
        //acceso al servidor

        for (int i = 1; i <= 10; i++) //se simulan 10 accesos al servidor
        {
            try {
                semaforo.acquire();
                //en cada acceso se adquiere el recurso
                //y si está ocupado se bloquea
            } catch (InterruptedException ex) {
            }
```

```
            servidor.incrementaCuenta();
            //adquirido el recurso, invoca a este método para simular el
acceso
            //al servidor incrementado la cuenta de accesos
            semaforo.release();
            //libera el recurso o permiso
                yield();
        }
    }
}
```

Clase Main:

```
import java.util.concurrent.Semaphore;

/**Se trata de simular el acceso simultáneo de 4 terminales (hilos) a un
servidor
 * y llevar la cuenta de accesos en cada instante. Desde cada terminal se
 * simularán 10 accesos.
 *
 * @author IMCG
 */
public class Main {

/**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
      Semaphore semaforo = new Semaphore(1);
       //semáforo para las secciones críticas de esta clase (permisos 1)
        ServidorWeb servidor = new ServidorWeb();
       //crea un objeto ServidorWeb
        Hilo_Terminal hterminal1 = new Hilo_Terminal(servidor, semaforo);
        Hilo_Terminal hterminal2 = new Hilo_Terminal(servidor, semaforo);
        Hilo_Terminal hterminal3 = new Hilo_Terminal(servidor, semaforo);
        Hilo_Terminal hterminal4 = new Hilo_Terminal(servidor, semaforo);
        //Se crean cuatro hilos

        hterminal1.start();
        hterminal2.start();
        hterminal3.start();
        hterminal4.start();
        //se inician los cuatro hilos
    }

}
```

Ejemplo 14 (UD2Ej14): Lectores-Escritores. Resuelto con semáforos de la clase Semaphore

Solución alternativa a este supuesto usando semáforos en vez de monitores

Clase Lector:

```
import java.util.concurrent.Semaphore;

public class Lector extends Thread {
//clase que implementa al hilo lector
    private Semaphore semaforo;

    //constructor
    public Lector(String nombre, Semaphore s) {
        super(nombre);
        this.semaforo = s;
    }

    public void run() {
        System.out.println(getName() + " : Intentando leer");
        //mensaje en consola para comprobar el funcionamiento
        try {
            semaforo.acquire();
            //solicita un permiso para acceder a la BD a leer
            //los otros 4 permisos, los pueden utilizar los otros hilos
lectores
        } catch (InterruptedException e) {
            System.out.println(e);
        }
        System.out.println(getName() + " : Leyendo");
        try {
            sleep((int) (Math.random() * 50));
            //se duerme al hilo un tiempo aleatorio (para simular que tarda
            //en realizar su tarea y así otros hilos compiten por el acceso
)
        } catch (InterruptedException e) {
            System.out.println(e);
        }

        System.out.println(getName() + " : Ya he leido");
        //mensaje en consola para comprobar el funcionamiento
        semaforo.release();
        //libera el permiso
    }
}
```

Clase Escritor:

```
import java.util.concurrent.Semaphore;

public class Escritor extends Thread {
//calse que implementa al hilo escritor
    private Semaphore semaforo;

    //constructor
    public Escritor(String nombre, Semaphore s) {
        super(nombre);
        this.semaforo = s;
    }
```

```
    @Override
    public void run() {
        System.out.println(getName() + " intentando escribir");
        //mensaje en consola para comprobar funcionamiento
        try {
            semaforo.acquire(5);
            //adquiere 5 permisos para asegurarse que la BD está libre
        } catch (InterruptedException e) {
            System.out.println(e);
        }
        System.out.println(getName() + ": Escribirndo");
        //mensaje en consola para comprobar el funcionamiento
        try {
            sleep((int) (Math.random() + 50));
            //se duerme al hilo un tiempo aleatorio (para simular que tarda
            //en realizar su tarea y así otros hilos compiten por el acceso
        } catch (InterruptedException e) {
            System.out.println(e);
        }


        System.out.println(getName() + ": Ya he escrito");
        //mensaje en consola para comprobar el funcionamiento
        semaforo.release(5);
        //libera los 5 permisos para indicar que la BD está libre
    }
}
```

Clase Main:

```
import java.util.concurrent.Semaphore;

public class Main {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
       Semaphore sema = new Semaphore(5);
       //semáforo que permite que un máximo de 5 hilos utilicen a la vez la
BD

        for(int i=1; i<=2;i++){
           new Escritor("Escritor " + i, sema).start();
           //crea e inicia 2 hilos escritores
        }
        for(int i=1; i<=5;i++){
           new Lector("Lector " + i, sema).start();
           //crea e inicia 5 hilos lectores
        }
        }
}
```

Ejemplo 15 (UD2Ej15): Productor-Consumidor. Sin sincronizar

En nuestro ejemplo, Productor y Consumidor comparten un Bufer; Productor escribe un dato en el Bufer, mientras que Consumidor lee un dato del Bufer

Lo primero a implementar en este ejemplo es el bufer mediante la clase Cola, que contienen dos métodos: put(int), que es usada por el Productor para asignar un valor a Bufer; y get(), usada por el Consumidor para obtener el valor de Bufer

El hilo productor se implementa con la clase Productor que se inicializa con un identificador numérico y el objeto cola sobre el que actúa. Realiza un bucle de 5 iteraciones en cada una de las cuales coloca un valor en la cola y luego se duerme 100 ms.

El hilo consumidor se implementa con la clase Consumidor que se inicializa con un identificador numérico y el objeto cola sobre el que actúa. Realiza un bucle de 5 iteraciones en cada una de las cuales extrae un valor en la cola y lo muestra por pantalla

La clase Main contiene la función principal donde se crea la cola, un productor y un consumidor y se lanza la ejecución de ambos

Clase Cola:

```
public class Cola {
	private int numero;
	private boolean disponible = false; //inicialmente cola vacia

	public int get() {
		if (disponible) { //hay numero en la cola?
			disponible = false; //se pone cola vacia
			return numero; //se devuelve
		}
		return (-1); //no hay numero disponible, cola vacia
	}

	public void put (int valor) {
		numero = valor; //coloca valor en la cola
		disponible = true; //disponible para consumir, cola llena
	}
}
```

Clase Productor:

```
public class Productor extends Thread {
	private Cola cola;
	private int n;

	public Productor (Cola c, int n){
		cola = c;
		this.n =n;
	}

	public void run() {
		for (int i = 0; i<5; i++) {
			cola.put(i); //pone el número
			System.out.println(i+"=>Productor : " + n + ", produce: "
+ i);
			try {
```

```
                        sleep(100);
                    } catch (InterruptedException e) { }
                }
        }
}
```

Clase Consumidor:

```
public class Consumidor extends Thread {
        private Cola cola;
        private int n;

        public Consumidor (Cola c, int n){
                cola = c;
                this.n =n;
        }

        public void run() {

                int valor = 0;
                for (int i = 0; i<5; i++) {
                        valor = cola.get(); //recoge el numero
                        System.out.println(i+"=>Consumidor : " + n + ", consume:
" + valor);
                }
        }
}
```

Clase Main:

```
public class Main {
        public static void main (String[] args) {
                Cola cola = new Cola();
                Productor p = new Productor(cola, 1);
                Consumidor c = new Consumidor(cola, 1);
                p.start();
                c.start();
        }
}
```

Al ejecutar este ejemplo se observará que el consumidor va más rápido que el productor (al que se le puso un sleep()) y no consume todos los números cuando se producen.

Para conseguir la salida deseada

```
0=>Productor1 : produce: 0
0=>Consumidor1 : consume: 0
1=>Productor1 : produce: 1
1=>Consumidor1 : consume: 1
2=>Productor1 : produce: 2
2=>Consumidor1 : consume: 2
3=>Productor1 : produce: 3
3=>Consumidor1 : consume: 3
4=>Productor1 : produce: 4
4=>Consumidor1 : consume: 4
```

Sería necesario que los hilos estuviesen sincronizados