Programación multihilo

Introducción a la programación multiproceso y multihilo

Multiproceso

El multiproceso consiste en la ejecución de varios procesos diferentes de forma *simultánea* para la realización de una o varias tareas relacionadas o no entre sí. En este caso, cada uno de estos procesos es una aplicación independiente. El caso más conocido es aquel en el que nos referimos al Sistema Operativo (Windows, Linux, MacOS, . . .) y decimos que es *multitarea* puesto que es capaz de ejecutar varias tareas o procesos (o programas) al mismo tiempo.

Multihilo

Hablamos de multihilo cuando se ejecutan varias tareas relacionadas o no entre sí dentro de una misma aplicación. En este caso no son procesos diferentes sino que dichas tareas se ejecutan dentro del mismo proceso del Sistema Operativo. A cada una de estas tareas se le conoce como hilo o thread (en algunos contextos también como procesos ligeros).

En ambos casos estaríamos hablando de lo que se conoce como **Programación Concurrente**. Hay que tener en cuenta que en ninguno de los dos casos la ejecución es realmente simultánea, ya que el Sistema Operativo es quién hace que parezca así, pero los ejecuta siguiendo lo que se conoce como *algoritmos de planificación*.

Algoritmos de planificación

En entornos multitarea, un **algoritmo de planificación** indica la forma en que el tiempo de procesamiento debe repartirse entre todas las tareas que deben ejecutarse en un

momento determinado. Existen diferentes algoritmos de planificación, cada uno con sus ventajas e inconvenientes, pero todos intentan cumplir con los siguientes puntos:

- Debe ser imparcial y eficiente
- Debe minimizar el tiempo de respuesta al usuario, sobre todo en aquellos procesos o tareas más interactivas
- Debe ejecutar el mayor número de procesos
- Debe mantener un equilibrio en el uso de los recursos del sistema

FCFS: First Come First Served

El primer proceso que llegue al procesador se ejecuta antes y de forma completa. Hasta que su ejecución no termina no podrá pasarse a ejecutar otro proceso.

RR: Round Robin

Se le conoce también como algoritmo de turno rotatorio. En este caso se designa una cantidad corta de tiempo (*quantum*) de procesamiento a todas las tareas. Las que necesiten más tiempo de proceso deberán esperar a que vuelva a ser su turno para seguir ejecutándose.

SPF: Shortest Process First

En este algoritmo, de todos los procesos listos para ser ejecutados, lo hará primero el más corto

SRT: Shortest Remaining Time

De todos los procesos listos para ejecución, se ejecutará aquel al que le quede menos tiempo para terminar.

Varias colas con realimentación

Es un algoritmo más complejo que todos los anteriores y, por tanto, más realista. Se utiliza en entornos donde se desconoce el tiempo de ejecución de un proceso al inicio de su ejecución. En este caso, el sistema dispone de varias colas que a su vez pueden disponer de diferentes políticas unas de otras. Los procesos van pasando de una cola a otra hasta que terminan su ejecución. En algunos casos, el algoritmo puede adaptarse modificando el número de colas, su política, . . .

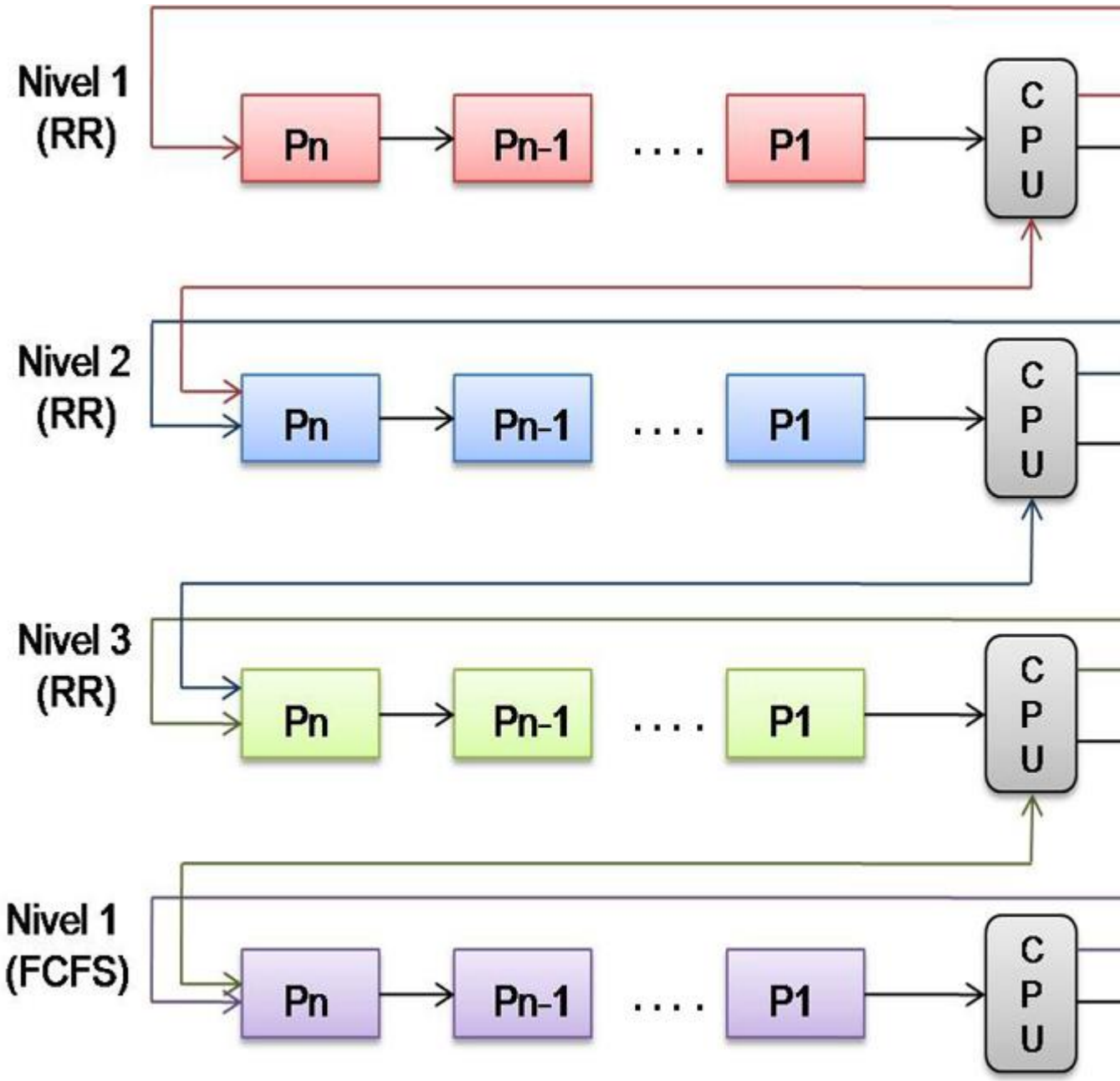


Figure 1: Algoritmo varias colas con realimentación

Ejercicios

1. Escribe un programa en Java que atienda una serie de tareas usando un algoritmo FCFS. Muestra las tareas que se van realizando en cada momento y el tiempo en que se inicia y termina cada una
2. Escribe ahora el mismo programa pero usando un algoritmo RR
3. Escribe ahora el mismo programa pero usando un algoritmo SRT

Programación concurrente, paralela y distribuida

Programación concurrente

Es la programación de aplicaciones capaces de realizar varias tareas de forma simultánea utilizando hilos o threads. En este caso todas las tareas compiten por el uso del procesador (lo más habitual es disponer sólo de uno) y en un instante determinado sólo una de ellas se encuentra en ejecución. Además, habrá que tener en cuenta que diferentes hilos pueden compartir información entre sí y eso complica mucho su programación y coordinación.

Programación paralela

Es la programación de aplicaciones que ejecutan tareas de forma paralela, de forma que no compiten por el procesador puesto que cada una de ellas se ejecuta en uno diferente. Normalmente buscan resultados comunes dividiendo el problema en varias tareas que se ejecutan al mismo tiempo.

Programación distribuida

Es la programación de aplicaciones en las que las tareas a ejecutar se reparten entre varios equipos diferentes (conectados en red, a los que llamaremos nodos). Juntos, estos equipos, forman lo que se conoce como un *Sistema Distribuido*, que busca formar redes de equipos que trabajen con un fin común [1)]

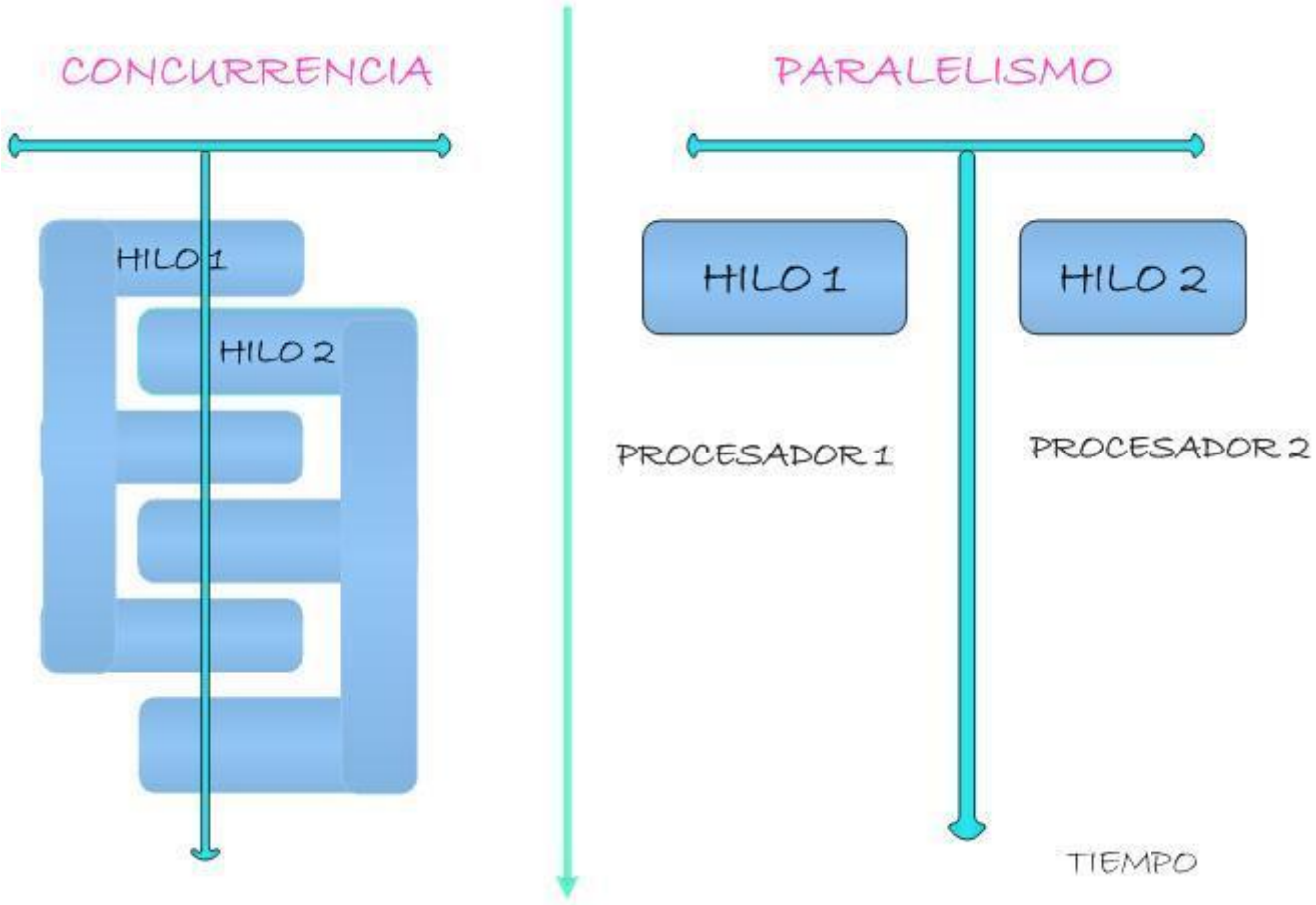


Figure 2: Programación concurrente / paralela

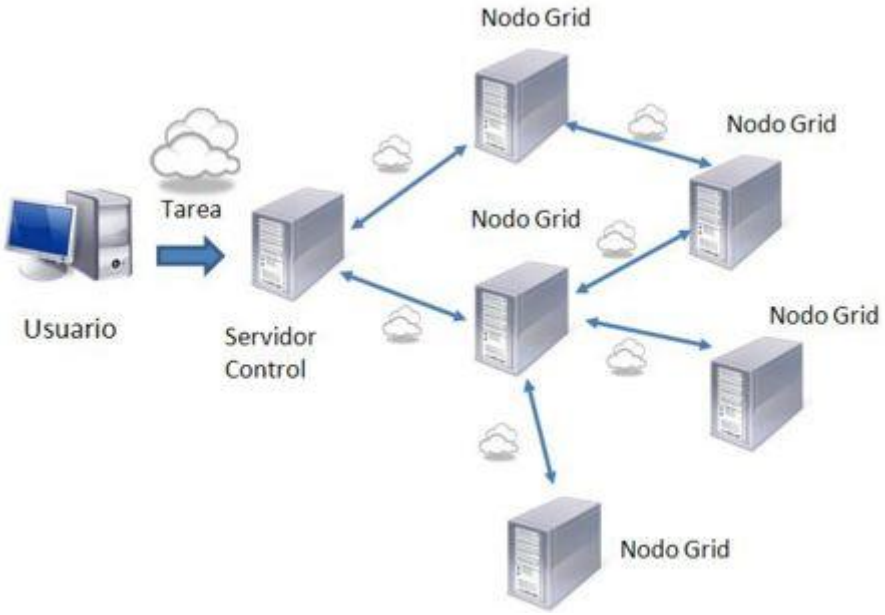


Figure 3: Programación distribuida

¿Qué son los hilos?

Un hilo o thread es cada una de las tareas que puede realizar de forma simultánea una aplicación. Por defecto, toda aplicación dispone de un único hilo de ejecución, al que se

conoce como *hilo principal*. Si dicha aplicación no despliega ningún otro hilo, sólo será capaz de ejecutar una tarea al mismo tiempo en ese hilo principal.

Así, para cada tarea adicional que se quiera ejecutar en esa aplicación, se deberá lanzar un nuevo hilo o thread. Para ello, todos los lenguajes de programación, como Java, disponen de una API para crear y trabajar con ellos.

En cualquier caso, es muy importante conocer los estados en los que se pueden encontrar un hilo. Estos estados se suelen representar mediante un gráfico como el que sigue:

Estados de un hilo

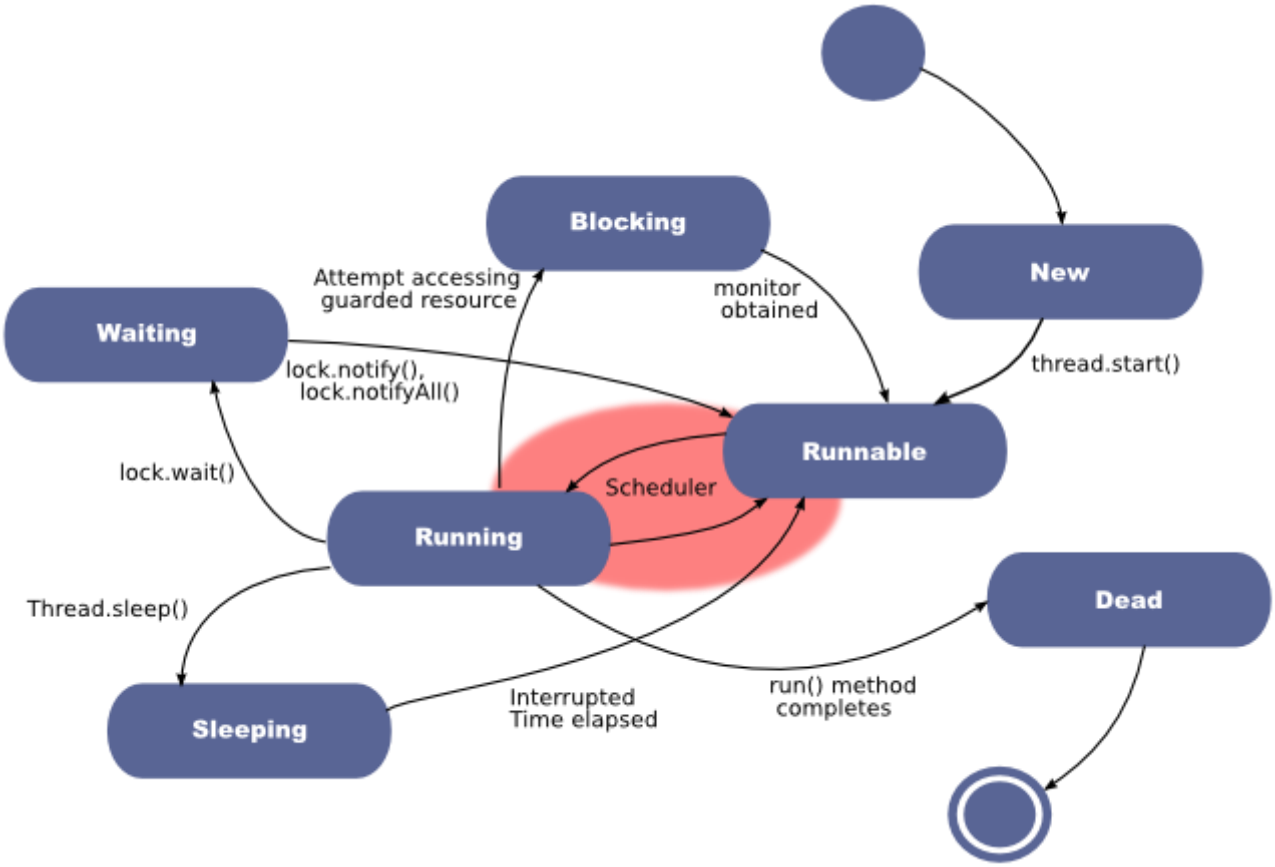


Figure 4: Estados de un hilo

Programación multihilo en Java

Creación y ejecución de un hilo

Para la creación de hilos en Java disponemos de varias vías, combinando el uso de la clase Thread y el interface Runnable según nos interese:

- Podemos utiliza la clase `Thread` heredando de ella. Es quizás la forma más cómoda porque una clase que hereda de `Thread` se convierte automáticamente en un hilo. Tiene una pega: esa clase ya no podrá heredera de ninguna otra, por lo que si la arquitectura de nuestra aplicación lo requiere ya no podríamos.
- Si tenemos la limitación que acabamos de comentar para el primer caso, podemos implementar el interface `Runnable` de forma que la clase que nosotros estamos

implementado podrá además heredar sin ninguna limitación. Sólo cambia un poco la forma de trabajar directamente con la clase hilo.

- Por otra parte también podemos crear un hilo utilizando una clase anónima. No es un método que se recomiende pero en algunos casos, cuando la clase que hace de hilo no va a tener una estructura concreta es bastante cómodo hacerlo de esta manera.

Crear un hilo heredando de la clase Thread

En este caso, la clase `Tarea` se conviene automáticamente en un hilo por el mero hecho de heredar de `Thread`. Sólo tenemos que tener en cuenta que, al heredar de esta clase, tenemos que implementar el método `run()` y escribir en él el código que queremos que esta clase ejecute cuando se lance como un hilo con el método `start()` (que también hereda de `Thread`)

```
public class Tarea extends Thread {

  @Override

  public void run() {

    for (int i = 0; i < 10; i++) {

      System.out.println("Soy un hilo y esto es lo que hago");

    }

  }

}



. . .



public class Programa {

  public static void main(String args[]) {

    Tarea tarea = new Tarea();

    tarea.start();

    System.out.println("Yo soy el hilo principal y sigo haciendo mi
trabajo");

    System.out.println("Fin del hilo principal");

  }
```

```
}
```

Crear un hilo implementado el interfaz Runnable

En este caso, suponemos que necesitamos que nuestra clase hilo herede de una segunda clase. En este caso, la clase deberá además implementar el interfaz `Runnable` y, como en el primer caso, implementar el método `run()` con la misma idea que en el punto anterior. Más adelante, tendremos que crear un objeto directamente de la clase `Thread` y pasarle como parámetro al constructor un objeto de nuestra clase hilo. De esa manera, el objeto `Thread` será un hilo que se comportará como el método `run()` de nuestra clase `Tarea` haya definido.

```
public class OtraClase {

  . . .

  . . .

}



. . .



public class Tarea extends OtraClase implements Runnable {

  @Override

  public void run() {

    for (int i = 0; i < 10; i++) {

      System.out.println("Soy un hilo y esto es lo que hago");

    }

  }

}



. . .
```

```
public class Programa {

  public static void main(String args[]) {

    Tarea tarea = new Tarea();

    Thread hilo = new Thread(tarea);

    hilo.start();

    System.out.println("Yo soy el hilo principal y sigo haciendo mi
trabajo");

    System.out.println("Fin del hilo principal");

  }

}
```

Crear un hilo implementado una clase anónima

Por último, implementar una clase anónima también permite crear hilos aunque sólo se recomienda para casos en los que la clase que se convierte en un hilo no tenga una estructura muy compleja ya que quedaría un código bastante ilegible.

```
public class Programa {

  public static void main(String args[]) {



    Thread hilo = new Thread(new Runnable() {

      @Override

      public void run() {

        for (int i = 0; i < 10; i++) {

        System.out.println("Soy un hilo y esto es lo que hago");

      }

    });
```

```
    hilo.start();

    System.out.println("Yo soy el hilo principal y sigo haciendo mi
trabajo");

    System.out.println("Fin del hilo principal");

  }

}
```

En cualquier caso tenemos que tener siempre en cuenta las siguientes consideraciones:

- Siempre se debe sobreescribir (`Override`) el método `run()` e implementar allí lo que tiene que hacer el hilo
- Podemos hacer que el hilo haga un número finito de cosas o bien que esté siempre en segundo plano (tendremos que asegurar que el método `run()` se ejecuta de forma continuada)(¿cómo se hace eso?)
- Los problemas vienen cuando existen varios hilos. Hay que tener en cuenta que pueden compartir datos y código y encontrarse en diferentes estados de ejecución
- La ejecución de nuestra aplicación será *thread-safe* si se puede garantizar una correcta manipulación de los datos que comparten los hilos de la aplicación sin resultados inesperados (más adelante veremos cómo)
- Además, en el caso de aplicaciones multihilo, también nos puede interesar sincronizar y comunicar unos hilos con otros

Ejercicios

1. ¿Qué pasa si ejecutas varias veces el código de los ejemplos? ¿Siempre ocurre lo mismo?
2. ¿Existe alguna manera de asegurar que todo se va a ejecutar en un orden concreto?

Sincronización de hilos

El API de Java proporciona una serie de métodos en la clase `Thread` para la sincronización de los hilos en una aplicación:

- `join()` Se espera la terminación del hilo que invoca a este método antes de continuar
- `Thread.sleep(int)` El hilo que ejecuta esta llamada permanece *dormido* durante el tiempo especificado como parámetro (en ms)
- `isAlive()` Comprueba si el hilo permanece activo todavía (no ha terminado su ejecución)
- `yield()` Sugiere al *scheduler* que sea otro hilo el que se ejecute (no se asegura)

También resulta interesante saber cómo detener un hilo. En este caso, la API de Java desaconsejó el método `stop()` que en un principio se ideó para detener la ejecución. Así, hoy en día, se nos anima a que seamos nosotros quienes implementemos formas *limpias* de detener nuestros hilos.

join() I

```java
public static void main(String args[]) {

  Hilo hilo1 = new Thread(new Tarea());

  Hilo hilo2 = new Thread(new Tarea());



  hilo1.start();

  hilo2.start();



  . . .

  . . .



  try {

    hilo1.join();

    hilo2.join();

  } catch (InterruptedException ie) {

    ie.printStackTrace();

  }



  System.out.println("Fin de la ejecución de los dos hilos");

}
```

El hilo principal espera a que ambos hilos se hayan ejecutado para continuar (o para lo que sea)

Ejercicios

1. Prueba a ejecutar este código sin las llamadas al método join() de ambos hilos (y comprueba el resultado)

---

join() II

```
public static void main(String args[]) {

  Hilo hilo1 = new Thread(new Tarea());

  Hilo hilo2 = new Thread(new Tarea());



  hilo1.start();

  hilo1.join();



  hilo2.start();

  hilo2.join();



  System.out.println("Fin de la ejecución de los dos hilos");

}
```

En este caso los hilos se ejecutan uno después de otro

Ejercicios

1. ¿Qué pasa si eliminamos las llamadas al método join() en ambos casos?
2. Prueba a hacerlo con más hilos. ¿Y con n hilos?

Thread.sleep

```
public class Tarea implements Runnable {

  @Override

  public void run() {

    for (int i = 0; i < 10; i++) {

      System.out.println("Soy un hilo y esto es lo que hago");

      try {

        Thread.sleep(500);

      } catch (InterruptedException ie) {

        ie.printStackTrace();

      }

    }

  }

}
```

En este caso el hilo *duerme* (detiene su ejecución) durante el tiempo especificado (en ms). Durante ese momento podrán ejecutarse otros hilos

isAlive()

```
public class TareaPrincipal implements Runnable {



  @Override

  public void run() {

    for (int i = 0; i < 10; i++) {

      System.out.println("Soy la TarePrincipal");

      try {

        Thread.sleep(500);
```

```
      } catch (InterruptedException ie) {
        ie.printStackTrace();
      }
    }
  }
}


. . .


public class TareaAlive implements Runnable {
  private Thread otroHilo;

  public TareaAlive(Thread otroHilo) {
    this.otroHilo = otroHilo;
  }

  @Override
  public void run() {
    while (otroHilo.isAlive()) {
      System.out.println("Yo hago cosas mientras el otro hilo siga en
ejecución");
      try {
        Thread.sleep(500);
      } catch (InterruptedException ie) {
```

```
        ie.printStackTrace();
      }
    }

    System.out.println("El otro hilo ha terminado. Yo también");
  }
}

. . .

public class Programa {
  public static void main(String args[]) {
    TarePrincipal tareaPrincipal = new TareaPrincipal();
    Thread hiloPrincipal = new Thread(tareaPrincipal);

    TareaAlive tareaAlive = new TareaAlive(hiloPrincipal);
    Thread hiloAlive = new Thread(tareaAlive);

    hiloPrincipal.start();
    hiloAlive.start();

    System.out.println("Se han terminado los dos hilos?");
  }
}
```

`isAlive()` está indicando que el hilo *está vivo* (ha iniciado su ejecución y aún no ha muerto, puede estar en cualquier estado intermedio, incluso durmiendo)

Ejercicios

1. ¿Podemos asegurar el resultado de la aplicación anterior?
2. ¿Y si lo hacemos extendiendo de la clase Thread a la hora de implementar los hilos?

Utilización de hilos en GUIs (Graphical User Interfaces)

En ocasiones, en aplicaciones con GUIs, el usuario debe esperar a que una tarea que se ejecuta en segundo plano termine. Mientras tanto, es muy habitual notificar el estado de la misma durante su ejecución o bien cuando termine. Esa notificación se puede hacer de muchas maneras, usando etiquetas de texto, barras de progreso, ventanas emergentes u otros recursos.

Puesto que Java "pinta" la GUI en la pantalla utilizando un hilo, necesitamos que todas las tareas que ejecutan operaciones sobre esa GUI lo hagan dentro del mismo hilo. Si dos hilos diferentes actúan sobre un mismo componente gráfico pueden darse situaciones no previstas. Es por eso que Java tiene en su API la clase `SwingWorker`, destinada para la creación de tareas en segundo plano que interactúan con el GUI de la aplicación. Así, cuando los hilos de la aplicación se ejecutan sin comportamientos inesperados podemos decir que lo hacen de forma *thread-safe*.

La clase SwingWorker

La clase `SwingWorker` pertenece al API de Java y permite ejecutar tareas sobre GUIs de forma *thread-safe*, permitiendo que éstas corran en el mismo hilo que el que Java hace funcionar para todos los controles *Swing* (*Event Dispatch Thread*).

En aplicaciones con GUIs varios hilos pueden actuar sobre el mismo elemento gráfico (una etiqueta de texto, por ejemplo) por lo que varios hilos compartirán ese recurso. Así, la clase `SwingWorker` garantiza que el acceso a ese recurso compartido se hace de forma *thread-safe*.

Desde el punto de vista de la implementación, además de codificar qué queremos que haga esta tarea, podremos devolver el resultado (si procede) y también notificar el avance de la misma (si procede, en una barra de progreso, por ejemplo). Además, también podremos implementar un `PropertyChangeListener` para actualizar el estado de la GUI de la aplicación mientras se ejecuta esta tarea en el segundo plano.

```
public class Tarea extends SwingWorker <ArrayList <BufferedImage>,
Integer> {
```

```
  private JProgressBar barraProgreso;

  // Se pasa como parametro la barra de progreso donde se quiere
notificar

  public Tarea(JProgressBar barraProgreso) {

    this.barraProgreso = barraProgreso;

  }



  @Override

  public ArrayList <BufferedImage > doInBackground() throws Exception {

    // Carga unas imagenes del disco duro a un Array y devuelve la lista

    . . .

    // Se notifica el avance (valor entre 0 y 100)

    setProgress(avanceCarga);

  }



  @Override

  public void process(List<Integer> valores) {

    // Aqui se reciben los valores del metodo publish()

    // Es una lista puesto que a vece se envian varios

    // en una sola llamada

    barraProgreso.setValue(valores.get(0));

  }

}
```

```
public class ProgramaGUI {

  . . .

  // Lanza la ejecucion de la tarea en segundo plano

  Tarea tarea = new Tarea(barraProgreso);

  tarea.execute();

  . . .

  // Obtiene el resultado de la carga de imagenes

  ArrayList<BufferedImage> resultado = tarea.get();

  . . .

}
```

Property Change Listener

```
public class Tarea extends SwingWorker <ArrayList <BufferedImage>,
Integer> {



  @Override

  public ArrayList <BufferedImage > doInBackground() throws Exception {

    // Carga unas imagenes del disco duro a un Array y devuelve la lista

    . . .

    // Se notifica el avance a traves del Property Change Listener

    setProgress(avanceCarga); }



  @Override

  public void done () {

    // Se notifica el final de la carga a traves del Property Change
Listener
```

```
    firePropertyChange("fin", false, true);
  }
}

public class ProgramaGUI {
  . . .
  Tarea tarea = new Tarea();
  tarea.addPropertyChangeListener(new PropertyChangeListener() {
    @Override
    public void propertyChange(PropertyChangeEvent event) {
      if (event.getPropertyName().equals("progress")) {
        int valor = (Integer) event.getNewValue();
        // Pintar este valor en barra de progreso o similar
      }
      else if (event.getPropertyName().equals("fin")) {
        // Fin de la carga de las imagenes
        // Hacer algo
      }
    }
  });
  tarea.execute();
  . . .
}
```

Patrón de diseño Observer

El patrón *Observer* es un patrón de diseño de software por el cual un objeto, llamada sujeto, mantiene a otros objetos llamados observadores a los cuales notifica cualquier cambio que sobre él se produce, normalmente invocando a alguno de sus métodos.
El API de Java incluye las clases necesarias para implementar este patrón, en el paquete `java.util`, con las clases Observable y Observer para definir al sujeto y a los observadores, respectivamente.

En el siguiente ejemplo se puede ver la implementación del patrón *Observer*. Tenemos dos clases, *Producto* y *Cliente*. La primera es el sujeto y la segunda es el observador de forma que cada vez que el producto cambie el precio (método `setStock(int)`) el cliente será notificado (mediante la llamada a `updateObservers()` que invoca el método `update()` en el cliente)

```
/**

 * Clase Producto

 * Es la clase Observable. Los cambios que se produzcan serán

 * observados inmediatamente por las clases Observer que se registren

 * desde ésta

 */

public class Producto extends Observable {



  private String nombre;

  private String descripcion;

  private float precio;

  private int stock;



  /*

   *  Único observador

   *  En el caso de que hubiera varios, podrían guardarse

   *  como colección
```

```
     */
    private Observer observer;

    public Producto() {}
    . . .
    // getters y setters para nombre, descripción y precio
    . . .

    public int getStock() { return stock; }

    public void setStock(int stock) {
      this.stock = stock;
      notifyObservers();
    }

    /**
     * Método que permite añadir observadores de esta clase
     */
    @Override
    public void addObserver(Observer observer) {
      this.observer = observer;
    }

    /**
```

```
    * Método que notifica los cambios a los observadores
    * de esta clase
    */
   @Override
   public void notifyObservers() {
     if (observer != null)
       observer.update(this, "stock");
   }
}
/**
 * Clase Cliente
 * En este caso es la clase que observa (Observer)
 * Observa los cambios que se produzcan en otra clase, a la que se conoce
como
 * Observable
 */
public class Cliente implements Observer {


   private String codigo;
   private String nombre;
   private String apellidos;
   private String email;
   private Date fechaNacimiento;
```

```
  public Cliente() {}

  . . .

  // getters y setters para codigo, nombre, apellidos

  // email y fechaNacimiento

  . . .



  /**

   * Método que se ejecuta cuando se producen cambios

   * en la clase observada

   */

  @Override

  public void update(Observable o, Object arg) {



    if (arg.equals("stock"))

      System.out.println("El cliente ha sido notificado de un cambio en
el stock");

  }

}

public static void main(String args[]) {



  Producto producto = new Producto();

  producto.setNombre("Patatas");

  producto.setDescripcion("Patatas fritas");

  producto.setPrecio(1.20f);
```

```
  producto.setStock(0);



  Cliente cliente = new Cliente();

  cliente.setCodigo("CLI0001");

  cliente.setNombre("Un");

  cliente.setApellidos("tipo");

  cliente.setEmail("un@tipo.com");

  cliente.setFechaNacimiento(new GregorianCalendar().getTime());



  /*

   * Se añaden observadores a la clase observable

   */

  producto.addObserver(cliente);

  /*

   * Un cambio en la clase observada hará "reaccionar" a la clase
observadora

   */

  producto.setStock(23);

}
```

Ejercicios

1. Escribe el proyecto del ejemplo anterior del patrón Observer. ¿Qué ocurre cada vez que se hace un cambio en el stock de un producto?
2. ¿Y si quiero que también ocurra algo cuando se modifique el precio? ¿Y si sólo quiero que ocurra cuando baja un tanto por ciento?
3. ¿Puedo hacer que más de un cliente sean notificados al mismo tiempo? Escribe un ejemplo

---

Ficheros de Registro (Logs)

A continuación se muestra, utilizando la librería log4j, un ejemplo de aplicación donde se realizan una serie de trazas a lo largo de su ejecución. La aplicación está compuesta por las dos clases que se muestran a continuación, con el objetivo de mostrar la traza cuando son varias clases las que ejecutan código.

```
public class Aplicacion {



  private static final Logger logger =
LogManager.getLogger(Aplicacion.class);



  public static void main(String args[]) {



    // Diferentes niveles de traza

    logger.trace("Aplicación iniciada");

    logger.error("Error de algo");

    logger.trace("Aplicación finalizada");

    logger.debug("Información para depurar");

    logger.warn("Esto es un aviso");



    OtraClase unObjeto = new OtraClase();

    unObjeto.unMetodo();



    try {

      // Forzamos una excepción para registrar su traza con log4j

      int x = 5 / 0;

    } catch (Exception e) {
```

```
      logger.trace("Se ha producido una excepción");



      // Almacena la traza de la excepción como String y lo registra con
log4j

      StringWriter sw = new StringWriter();

      e.printStackTrace(new PrintWriter(sw));

      logger.error(sw.toString());

      }

  }

}

public class OtraClase {



  private static final Logger logger =
LogManager.getLogger(OtraClase.class);



  public void unMetodo() {



    logger.trace("Se ha ejecutado el método unMetodo");

  }

}
```

Antes de poder ejecutar la aplicación, se ha creado un fichero mínimo de configuración para *log4j* creando el siguiente fichero *XML* donde se habilita la traza por Consola y Fichero con un patrón de mensaje determinado

log4j2.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<Configuration status="WARN">
```

```
  <Appenders>

    <Console name="Console" target="SYSTEM_OUT">

      <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level
%logger{36} - %msg%n"/>

    </Console>

    <File name="Fichero" fileName="ejemplolog4j.log"
bufferedIO="false" advertiseURI="file://ejemplolog4j.log"
advertise="true">

      <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level
%logger{36} - %msg%n"/>

    </File>

  </Appenders>

  <Loggers>

    <Root level="trace">

      <AppenderRef ref="Console"/>

      <AppenderRef ref="Fichero"/>

    </Root>

  </Loggers>

</Configuration>
```

Así, para el ejemplo anterior, la traza resultante (tanto para consola como para el fichero `ejemplolog4j.log`) sería la siguiente

```
10:46:27.049 [main] TRACE com.sfaci.ejemplolog4j.Aplicacion - Aplicación
iniciada

10:46:27.051 [main] ERROR com.sfaci.ejemplolog4j.Aplicacion - Error de
algo

10:46:27.051 [main] TRACE com.sfaci.ejemplolog4j.Aplicacion - Aplicación
finalizada
```

```
10:46:27.051 [main] DEBUG com.sfaci.ejemplolog4j.Aplicacion - Información
para depurar

10:46:27.051 [main] WARN  com.sfaci.ejemplolog4j.Aplicacion - Esto es un
aviso

10:46:27.051 [main] TRACE com.sfaci.ejemplolog4j.OtraClase - Se ha
ejecutado el método unMetodo

10:46:27.052 [main] TRACE com.sfaci.ejemplolog4j.Aplicacion - Se ha
producido una excepción

10:46:27.052 [main] ERROR com.sfaci.ejemplolog4j.Aplicacion -
java.lang.ArithmeticException: / by zero

        at com.sfaci.ejemplolog4j.Aplicacion.main(Aplicacion.java:38)

        at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)

        at
sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java
:62)

        at
sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorI
mpl.java:43)

        at java.lang.reflect.Method.invoke(Method.java:483)

        at
com.intellij.rt.execution.application.AppMain.main(AppMain.java:134)
```

Ejercicios

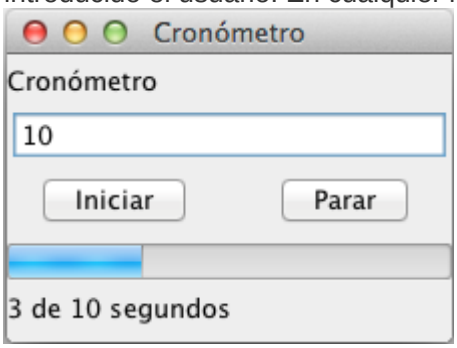
1. Realiza una aplicación de consola que cuente hasta un número determinado (mostrando la secuencia por pantalla) utilizando dos hilos, de forma que cada uno de ellos cuente un rango de números

2. Realiza una aplicación de consola que cuente hasta un número determinado (mostrando la secuencia por pantalla) utilizando un número determinado de hilos. La secuencia de números se repartirá a partes iguales entre todos los hilos de forma que a cada uno se le asigne un rango

3. Realiza una aplicación que simule una carrera de coches (de hasta 4 coches). Para cada coche se podrá configurar su velocidad y en la aplicación podremos configurar la distancia del circuito. Una vez lanzada la carrera se irá mostrando por pantalla (mediante barras de progreso, por ejemplo) el desarrollo de la misma (el avance de cada coche en el tiempo). Al final de la carrera se anunciará el coche ganador y los demás se detendrán mostrando cuánta distancia han recorrido

4. Realiza una aplicación en la que el usuario pueda programar una cuenta atrás que al terminar muestre un mensaje en la pantalla principal. Además, se mostrará en una barra de progreso el transcurso de dicha cuenta atrás (vaciando la barra de progreso)

5. Realiza una aplicación en la que se muestre, mediante una barra de progreso y una etiqueta de texto, el tiempo que pasa, en segundos, hasta una cantidad que habrá introducido el usuario. En cualquier momento éste podrá cancelar la cuenta.



6. Realiza una aplicación que simule la persecución de dos aviones. Considerando los parámetros altura, velocidad y dirección, el avión perseguido puede trazar la ruta que quiera y cambiar su velocidad y altura y el avión perseguidor deberá modificar esos parámetros en la misma medida en que lo haga el primero. Puedes solicitar por consola cambiar los parámetros del avión perseguido o bien preparar un recorrido ya definido y lanzar la aplicación para ver como lo recorren ambos objetos (mostrando en cada instante los parámetros de cada uno)

7. Realiza una aplicación que descargue un fichero de Internet mostrando, al final, la duración de la descarga formato MM:SS

El proyecto lo tenéis dentro de la sección ejercicios llamado psp-ejercicios-master.zip