Capítulo XXI: Concurrencia

Motivación

Hasta ahora hemos visto que los programas corren y ejecutan en forma lineal algo. Por ejemplo, si estás ejecutando un problema matemático de integración, la solución es bastante pautada:

- Dibujar los rectángulos bajo la cubra
- Calcular el área de cada uno de los rectángulos
- Sumar las áreas
- Devolver como aproximación la suma

¿Qué significa esto? Pues que antes de poder continuar con el siguiente paso debes completar el paso actual. Una verdadera receta.

Pensemos un momento en el problema de un juego de damas.

- Cada pieza en el tablero posee un árbol de decisión (a veces 2 movidas, en caso de las piezas normales y más de 2 movidas en caso de las damas).
- El jugador posee entonces un árbol de decisión con al menos N subárboles, uno para cada pieza que tenga el tablero.

¿Creen que en forma lineal esto se resolvería rápidamente?

La respuesta a esta pregunta es que por supuesto que no. Si tenemos 10 piezas y solo analizamos un nivel, debemos considerar tantas movidas que ni siquiera podríamos mentalmente visualizarlas todas para elegir la mejor. Complicado sería para el computador hacerlo en tiempos razonablemente cortos.

Pensemos ahora en otro tipo de juegos. Imagínense que ustedes son los defensores de la tierra y vienen unos marcianos a atacarla con naves (¿Space Invaders?).

¿Cómo el computador va a manejar los movimientos de tu nave y los movimientos de los malos al mismo tiempo sin quedarse pegado?

Imagínate que se tarda 0.5 segundos en recoger un movimiento y realizarlo en pantalla. Además tenemos 25 naves en la pantalla dibujadas y que deben tener mov imiento propio. Por lo tanto de manera secuencial, dibujar la pantalla costaría una espera de 0.5 x 25 segundos, es decir, 12.5 segundos. Ahora llevamos nuestro juego al procesador más rápido del mundo que tarda solo 0.1 segundos en recoger y dibujar. Tomaría exactamente 2.5 segundos en toda la pantalla. Todo esto sin considerar que también se deben mover los proyectiles disparados de cada nave… ¡Ufff!... Me aburrí muy rápido de jugar.

Pero en realidad no hay tanto problema, porque la solución existe y se llama Concurrencia.

Conceptos

> *La Concurrencia es la capacidad de ejecutar distintos procesos de manera síncrona y de forma independiente.*

Hasta ahora nuestros programas no han tenido concurrencia, por supuesto. Con este concepto se acabaron los procesos batch y las largas esperas de procesamiento mientras se ejecuta el programa. Pero sin embargo, la concurrencia necesita de algo que se llama Proceso.

> *Un Proceso es un trozo de programa que se ejecuta en un espacio separado de memoria, aislado del programa principal.*

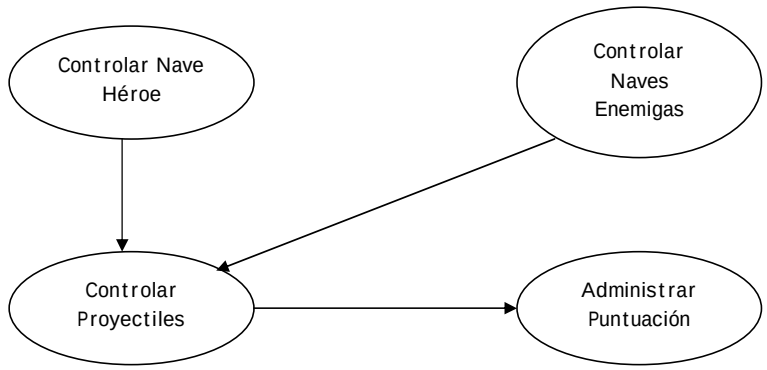
Básicamente un proceso es un programa en ejecución, pero que no depende de nadie. Este tan independiente programa ejecuta sus funciones y al terminar solo desaparece. El ejemplo más claro de estas cosas es el MSN Messenger. Si tienes tu Messenger ejecutándose como un Proceso, te permitirá navegar en internet, jugar Quake, hacer tu tarea de computa, pero sin interrumpir tu trabajo el continúa su ejecución.

¿Para qué sirven?

Imagínate nuevamente el Space Invaders que debe realizar muchas cosas:

- Mover naves enemigas
- Recibir movimientos de la nave héroe
- Mover proyectiles
- Calcular puntuación y manejar vidas

Si todo esto lo realiza en forma lineal, sería muy lento. Veamos cómo sería en forma paralela:



Las burbujas del diagrama representan los proceso que realiza y cómo se comunican (a grandes rasgos). Entonces, ¿cómo leer el diagrama?: "Mientras se controla a la nave héroe y a las naves enemigas, se verifica que los disparos lleguen a sus objetivos y si es así hay que sumar puntos o restar vidas.

Suena sencillo, pero en la práctica ¿cómo se hace?, con objetos especiales que veremos ahora:

Sintaxis

En concurrencia se utiliza una clase especial llamada Thread. Esta clase está construida de la siguiente forma:

```
public class java.lang.Thread
       extends java.lang.Object
       implements java.lang.Runnable {
    java.util.Map threadLocals;
    java.util.Map inheritableThreadLocals;
    public static final int MIN_PRIORITY;
    public static final int NORM_PRIORITY;
    public static final int MAX_PRIORITY;
    public static native java.lang.Thread currentThread();
    public static native void yield();
    public static native void sleep(long)
      throws java.lang.InterruptedException;
    public static void sleep(long, int)
      throws java.lang.InterruptedException;
    public java.lang.Thread();
    public java.lang.Thread(java.lang.Runnable);
    public java.lang.Thread(java.lang.ThreadGroup,
                            java.lang.Runnable);
    public java.lang.Thread(java.lang.String);
    public java.lang.Thread(java.lang.ThreadGroup,java.lang.String);
    public java.lang.Thread(java.lang.Runnable,java.lang.String);
    public java.lang.Thread(java.lang.ThreadGroup,
                            java.lang.Runnable,java.lang.String);
    public native synchronized void start();
    public void run();
    public final void stop();
    public final synchronized void stop(java.lang.Throwable);
    public void interrupt();
    public static boolean interrupted();
    public boolean isInterrupted();
    public void destroy();
    public final native boolean isAlive();
    public final void suspend();
    public final void resume();
    public final void setPriority(int);
    public final int getPriority();
    public final void setName(java.lang.String);
    public final java.lang.String getName();
    public final java.lang.ThreadGroup getThreadGroup();
    public static int activeCount();
    public static int enumerate(java.lang.Thread[]);
    public native int countStackFrames();
    public final synchronized void join(long)
      throws java.lang.InterruptedException;
    public final synchronized void join(long, int)
      throws java.lang.InterruptedException;
    public final void join() throws java.lang.InterruptedException;
```



```
    public static void dumpStack();
    public final void setDaemon(boolean);
    public final boolean isDaemon();
    public final void checkAccess();
    public java.lang.String toString();
    public java.lang.ClassLoader getContextClassLoader();
    public void setContextClassLoader(java.lang.ClassLoader);
    static {};
}
```

¿Qué significa esto? Bueno, en realidad son muchos métodos que se pueden utilizar con threads, pero que solo utilizaremos una pequeña parte en las clases que heredaremos de ella.

Para crear un proceso independiente, se debe crear una clase que extienda de Thread:

```
public class MiThread extends Thread {
      public void run() {
            for (int i=1; i<=10; i++) {
                  System.out.println(i);
            }
            System.out.println("FIN DEL THREAD");
      }
}
```

Este thread como pueden verlo posee solo un método propio (todos los demás los trae la clase Thread) y ese se llama run(). Este método es OBLIGATORIO (al igual que el main en el caso de los programas normales) y es el encargado de ejecutar el thread, es decir, allí se programa lo que el thread hace. En este caso, solo imprime los números del 1 al 10 en pantalla, uno por línea.

Si nosotros compilamos esta clase y la ejecutamos no pasaría nada, de hecho echaría de menos el main el compilador y reclamaría. ¿Por qué?, bueno, porque no se puede ejecutar un thread como si fuese un programa principal, si no que se utiliza lanzándolo desde otro programa:

```
public class programa {
      static public void main (String[] args) {
            Thread t = new MiThread();
            t.start();
            System.out.println("FIN DEL PROGRAMA PRINCIPAL");
      }
}
```

Notemos claramente que este programa crea un thread del tipo personalizado que cuenta de 1 a 10 en pantalla (clase MiThread), lo echa a correr utilizando el método start() nativo de la clase Thread y luego pone en pantalla el texto "FIN DEL PROGRAMA PRINCIPAL". Con lo que sabemos ahora, esperaríamos la siguiente salida:

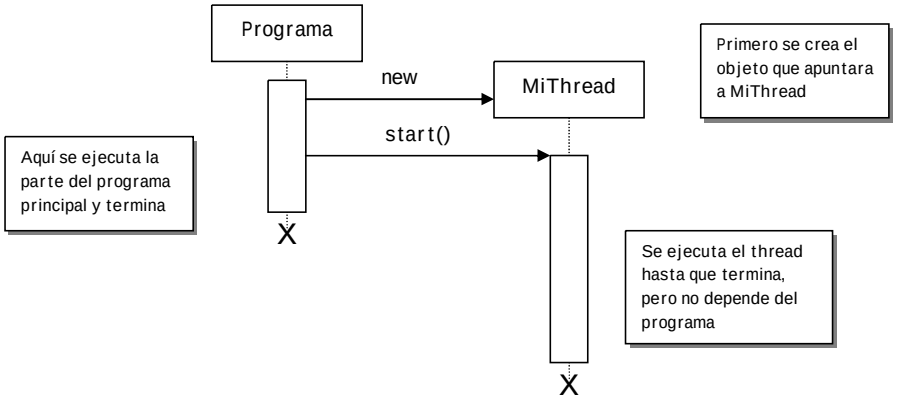
```
1
2
...
9
10
FIN DEL THREAD
FIN DEL PROGRAMA PRINCIPAL
```

Sin embargo esto no es realmente lo que pasa, y la salida real es la siguiente:

```
FIN DEL PROGRAMA PRINCIPAL
1
2
...
9
10
FIN DEL THREAD
```

¿Que diablos?

Bueno, este tema es sencillo. Veamos con un diagrama de proceso lo que ocurre (ojo que se parece a un diagrama de interacción de UML):



Si se dan cuenta, el objeto Programa termina mucho antes que el objeto MiThread, es decir el programa puede terminar sin que el thread haya terminado su labor. Línea a línea pasaría lo siguiente:

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

        Thread t = new MiThread();

        t.start();

        System.out.println("...");

}
}
```

```
public class MiThread extends Thread {

public void run() {

        System.out.println("1");

        System.out.println("2");
        ...
        System.out.println("9");
        System.out.println("10");

        System.out.println("...");

}
}
```

Entonces, el thread comienza justo al hacer el start().

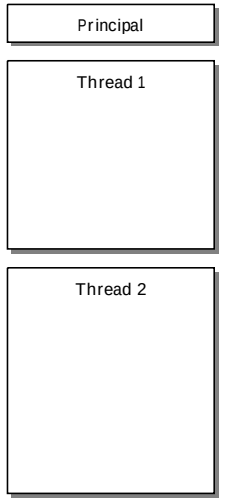
Compliquemos el ejemplo. Supongamos que nuestro programa principal sea:

```
public class programa {
      static public void main (String[] args) {
            Thread t1 = new MiThread();
            Thread t2 = new MiThread();
            t1.start();
            t2.start();
            System.out.println("FIN DEL PROGRAMA PRINCIPAL");
      }
}
```

¿Qué debería salir?

En este caso no es facil adivinarlo, porque la ejecución de 2 thread se torna compleja. Así que la salida al ejecutar el programa es:

```
FIN DEL PROGRAMA PRINCIPAL
1
2
3
4
5
6
7
8
9
10
FIN DEL THREAD
1
2
3
4
5
6
7
8
9
10
FIN DEL THREAD
```



¡Hey, eso sí que es extraño!

Si lo pensamos de cómo funciona Java, pues no es tan extraño. Recuerda que el thread posee el método start() definido como synchronized, es decir, se ejecuta un método run() a la vez. Pero entonces, ¿dónde está el paralelismo? ¡esto es secuencial!. Veamos si cambiamos algo en MiThread:

```
public class MiThread extends Thread {
      public void run() {
            for (int i=1; i<=10; i++) {
                  System.out.println(i);
                  try {
                        super.sleep(10);
                  }
                  catch (InterruptedException e) {
```

```
                }
            }
            System.out.println("FIN DEL THREAD");
        }
}
```

Y la nueva salida sería:

```
FIN DEL PROGRAMA PRINCIPAL
1
1
2
2
3
3
4
4
5
5
6
6
7
7
8
8
9
9
10
10
FIN DEL THREAD
FIN DEL THREAD
```

¡Ahora si que quedamos mal!....

Si tampoco es tan complicado el tema si explicamos primero cuál fue la modificación que le hicimos al thread para que hiciera esto. sleep(long) es un método de la clase Thread que permite hacer "dormir" el proceso durante una cantidad de milisegundos (¡si!, una pequeña siesta) que se le pasan por parámetros, es decir, que se queda "esperando" durante un momento antes de continuar con la ejecución.

Como en este caso estamos esperan 10 milisengundos, podemos decir que se intercala la ejecución de cada thread, mostrando que quedan prácticamente en paralelo (los dos al mismo tiempo).

Además, si sumamos el hecho que el start() estaba sincronizado, significa que cada proceso hará un ciclo por vez en el procesador, y cada vez que el proceso haga sleep() estaremos avisando para que otro tome el control del procesador por un ciclo más o hasta que éste haga otro sleep. Cuando el sleep se acaba, el proceso que despierta toma el control y comienza de nuevo a ejecutar desde donde se durmió.

Otra forma de “soltar” el procesador para que otro thread pase a trabajar en él es usando el método yield() en vez de usar sleep(). En este caso, yield() permite que el proceso desocupe el procesador sin pasar a estado inactivo.

Manejando las Excepciones del Thread

Como lo dice su definición, algunos métodos deben manipular excepciones. ¿Cuáles son esas excepciones?, bueno, eso depende del método.

En el caso del método run() estamos obligados a capturar la excepción InterruptedException que nos permite controlar cuando el thread es interrumpido en su ejecución. Esta interrupción puede pasar por distintas razones: falta de memoria, llamada del proceso, etc.

Ejemplo

Veamos ahora un programa "real". Queremos hacer la gráfica de un reloj, pero que se mueva segundo a segundo un indicador. Sería sencillo hacer primero la gráfica:

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class Reloj {
  private Frame programa;
  private Canvas area;

  public Reloj() {
    programa = new Frame("JReloj");
    programa.setLayout(new FlowLayout());

    area = new Canvas();
    area.setSize(600, 600);
    programa.add(area);

    programa.pack();
    programa.show();

    Graphics pincel = area.getGraphics();
    pincel.setColor(Color.blue);
    pincel.drawOval(10, 10, 590, 590);

    Thread seg = new Timer(pincel, 300, 200);
    seg.start();
  }

  public static void main(String[] args) {
    Reloj r = new Reloj();
  }
}
```

Fijémonos que la lógica de la interfaz es nula y que solo estamos pasándole el control del reloj a un proceso Timer llamado seg. La lógica entonces estaría acá:

```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

class Timer extends Thread {
       Graphics segundero;
       int pos_x, pos_y;
       long seg = 180;
       public Timer (Graphics g, int x, int y) {
```

```
            this.segundero = g;
            this.pos_x = x;
            this.pos_y = y;
      }

      public void run() {
            int ang = 6, x, y;
            while(true) {
                  x = (int) Math.round(250 *
                        Math.sin(Math.toRadians(seg)));
                  y = (int) Math.round(250 *
                        Math.cos(Math.toRadians(seg)));

                  segundero.setColor(Color.blue);
                  segundero.drawLine(pos_x,  pox_y,
                        pos_x + x, pos_y + y);

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

                  segundero.setColor(Color.white);
                  segundero.drawLine(pos_x,  pox_y,
                        pos_x + x, pos_y + y);

                  seg -= ang;
                  if (seg == 360) {
                        seg = 0;
                  }
            }
      }
}
```

En el constructor hacemos almacenar la posición inicial del segundero y además el pincel que nos permitirá dibujar sobre el canvas de la interfaz creada antes. El método run() en este caso tiene la misma estructura que cualquiera y se duerme el proceso cada cierto tiempo por un lapso de 1 segundo. Mientras el proceso se despierta, borra el segundero desde donde estaba, le aumenta el ángulo al segundero (en este caso el ángulo es 6, porque si dividimos 360 / 60 partes nos queda que cada 6º debería correr el segundero) y luego calculamos las nuevos valores para la proyección sobre el eje x e y del segundero y así dibujarlo. Luego de esto, a dormir de nuevo.

La lógica es super simple, pero el programa ya se ve más complicado. Sin embargo la parte de paralelismo o concurrencia es siempre igual.

Sin embargo, la gracia del ejemplo radica en la sencilla premisa de que el procesador se mantiene más tiempo desocupado que si lo hubiésemos hecho dentro de la clase de la interfaz. Esto implica:

- Ahorro de memoria
- Ahorro de tiempo de espera
- Aprovechamiento del tiempo ocioso del procesador

Más Métodos de Threads

Hasta ahora, los programas que hemos visto solo permiten lanzar procesos, los cuales comienzan a funcionar en forma independiente del proceso principal. Existen veces en que esos programas necesitan tener conciencia de qué están haciendo los procesos que han lanzado y poder controlar tanto el tiempo de vida como el tiempo que pasan en estado inactivo (durmiendo).

join() es un método que permite al programa que lanza un proceso, entregarle el control del procesador durante el tiempo que este viva, y el programa principal esperará hasta que el proceso termine para continuar. Por ejemplo, si tenemos el mismo programa que cuenta de 1 a 10:

```
public class MiThread extends Thread {
      public void run() {
            for (int i=1; i<=10; i++) {
                  System.out.println(i);
            }
            System.out.println("FIN DEL THREAD");
      }
}
```

```
public class Programa {
      static public void main (String[] args) {
            Thread t = new MiThread();
            t.start();
            System.out.println("FIN DEL PROGRAMA PRINCIPAL");
      }
}
```

La salida que obtenemos es:

```
FIN DEL PROGRAMA PRINCIPAL
1
2
...
9
10
FIN DEL THREAD
```

Lógicamente no es muy adecuado, ya que tendríamos la esperanza que el texto "FIN DEL PROGRAMA PRINCIPAL" apareciece al final. Para ello, usamos el join():

```
public class MiThread extends Thread {
      public void run() {
            for (int i=1; i<=10; i++) {
                  System.out.println(i);
            }
            System.out.println("FIN DEL THREAD");
      }
}
```

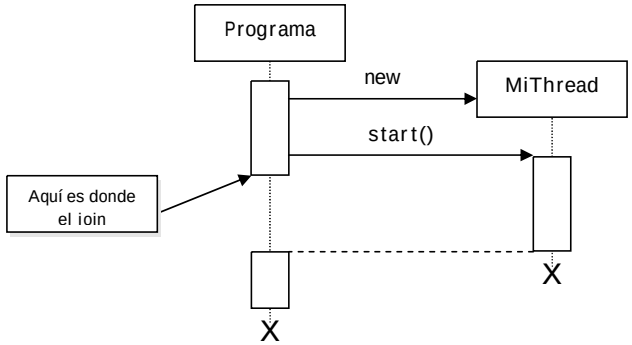
```
public class Programa {
      static public void main (String[] args) {
            Thread t = new MiThread();
```

```
            t.start();
            t.join();
            System.out.println("FIN DEL PROGRAMA PRINCIPAL");
      }
}
```

Lo que ahora si produciría:

```
1
2
...
9
10
FIN DEL THREAD
FIN DEL PROGRAMA PRINCIPAL
```

La traza del programa sería:



Por otro lado, también es posible interrumpir el proceso en cualquiera de sus estados. Para ello se debe utilizar el método interrupt() que puede causar distintos efectos:

- Si el proceso no permite accesos, el thread recibe una SecurityException.
- Si el proceso se encuentra dormido (sleep) o en espera (wait o join), el thread recibe una InterruptedException.
- Si el proceso está corriendo actualmente, se deja en estado interrumpido, y el thread puede verificar ese estado consultando el método isInterrupted() el cual retorna TRUE si el thread está interrumpido.

De esta forma, el método puede saber cuando es interrumpido en cualquier momento.

Prioridad de un Proceso

Recordemos que, dependiendo de los "start" que uno realice, los proceso van quedando "encolados" para ser ejecutados por el procesador. Pero existen siempre posibilidad de ordenar los procesos según algún parámetro obligado, al que llamaremos prioridad. En este caso no es más que un número entero que indica qué proceso se debe ejecutar primero que otro (entre mayor es la prioridad, prevalece antes que los de menor prioridad).

Para darle la prioridad a un proceso, se debe usar el método setPriority(int n), en el cual se indica que prioridad debe tener el proceso. Por defecto, cuando se crea el proceso la prioridad queda seteada en Thread.NORM_PRIORITY (un valor estándar) y este valor debe ir entre Thread.MIN_PRIORITYy Thread.MAX_PRIORITY.

En el siguiente ejemplo se crean tres procesos distintos, los cuales son ejecutado en forma diferente a cómo fueron creados:

```
Thread t1 = new MiThread();
Thread t2 = new MiThread();
Thread t3 = new MiThread();
t3.setPriotity(t3.getPriority()+1);
t1.start();
t2.start();
t3.start();
```

Observando este caso, el órden de ejecución será realmente t1, t3 y luego t2, ya que t3 posee mayor prioridad que t2. Por otro lado, se usó el método getPriority() para saber la prioridad normal. También es posible usar las variables estáticas de la clase Thread:

```
Thread t1 = new MiThread();
Thread t2 = new MiThread();
Thread t3 = new MiThread();
t3.setPriotity(Thread.NORM_PRIORITY+1);
t1.start();
t2.start();
t3.start();
```

Procesos Sincrónicos

Hemos visto que los procesos pueden funcionar de una forma libre utilizando el o los procesadores del computador en forma paralela. Esto no causa ningún problema si la información a la cual están accediendo no es compartida entre los procesos. Pero, ¿qué pasa si los procesos deben acceder a la misma información?.

Por ejemplo, si se desea modelar el comportamiento de las cajas de un banco, utilizando procesos paralelos, debemos considerar que se debe acceder a las cuentas y, si no tenemos cuidado, podríamos estar accediendo con 2 cajas a la misma cuenta arriesgándonos a que ocurra una inconsistencia de datos.

Para ello, es necesario que el acceso a las cuentas esté "sincronizado". Dado este requerimiento, debemos sincronizar los métodos de acceso a las cuentas y no los procesos:

```
public class Cuenta {
      // Definición interna de la cuenta

      public Cuenta (String numero) { ... }
      public synchronized boolean giro(long monto) { ... }
      public synchronized void deposito(long deposito) { ... }
      public long saldo() { ... }
}
```

Con la clase Cuenta se modela la representación lógica de una cuenta del banco, en la cual se pueden hacer funciones de giro, depósito y saldo. El constructor permite obtener la información de la cuenta desde la "Base de Datos" de cuentas (que en este momento no es necesario saber cuál es).

En este caso solo hemos considerado como sincronizables aquellos métodos que podrían traer inconsistencias al ser concurrentes. En el caso de saldo, si dos cajeros consultan el saldo, no es necesario, ya que ambos procesos obtendrán el mismo saldo.

Considerando esta clase, podemos usar el proceso Cajero el cual realiza el modelo de llamar a la cuenta. En este caso, no hay problemas, ya que cuando el cajero realice una operación:

```
...
cta.giro(10000);
...
```

En este caso estaremos asegurándonos que solo ese proceso hará el giro solicitado, encolando a los siguientes procesos que llamen el método para acceder a las cuentas.

Monitor de Procesos

Otra forma de sincronizar los procesos es utilizando métodos nativos de Object. Estos métodos permiten que los procesos posean un órden, como si fuese una línea de producción (ensamblaje, pintura, control de calidad, etc).

Para realizar esto, poseemos 2 métodos básicos:

- **ü** wait(): Permite al proceso pasar a un estado inactivo mientras se ejecutan otros procesos. El proceso se mantendrá en ese estado hasta que alguien realice un notify() sobre él.
- **ü** notify(): Permite despertar a un proceso que se encuentre en estado inactivo a causa de un wait().

Ambos métodos requieren de controlar las InterruptedException, al igual que se hace con sleep(), y deben ser invocadas desde el programa que tenga el control del proceso, ya sea porque es el invocador del mismo, o posea acceso al objeto que está ejecutando el proceso en cuestión (Monitor de Procesos).

También existe notifyAll() que permite despertar a todos los procesos que estén esperando en el monitor de procesos.