élogos

APLICACIONES DE LA MULTITAREA

élogos

ÍNDICE

APLICACIONES DE LA MULTITAREA

1. Sincronización de hilos Java

Si se desea que dos hilos se comuniquen para compartir una estructura de datos compleja (como una lista enlazada), se necesita alguna manera de garantizar que cada uno se aparta del camino del otro.

Java incorpora una versión rebuscada de un modelo clásico para la sincronización, el monitor.

La mayor parte de los sistemas multihilo implementan los monitores a modo de objetos, pero Java proporciona una solución más elegante: no existe la clase monitor, cada objeto lleva asociado su propio monitor implícito, en el que puede entrar sin más sólo haciendo una llamada a los métodos synchronized del objeto.

Una vez que el hilo está dentro del método synchronized, ningún otro hilo puede efectuar una llamada a otro método synchronized sobre el mismo objeto.

La sección crítica y el mutex/monitor /Mutual Exclusion

La sección crítica es un dato (variable) o bien un trozo de programa que sólo puede ser usado por un programa a la vez. Un ente de programación que sólo puede ser accedido por un hilo a la vez.

Para apoderarnos del **mutex** llamamos a la función con el método **synchronized**

El primero en llamar al método le asigna el mutex de forma que si luego otro ejecuta la función synchronized, se la duerme hasta que el que tiene el mutex finaliza.

```
Synchronized <Tipo_Retorno> <Nombre_Método> (<Parámetros>) { ... }
```

Ejemplo

Aplicaciones de la multitarea

```
class Impresora {
// Se le pone synchronized y se ejecuta cada hilo por separado.
// Si no se le pone se mezclan las cadenas en la ejecución.
//Imprime carácter a carácter
synchronized void imprimir (String cadena) {
for (int i=0; i < cadena.length(); i++)
System.out.print(cadena.charAt(i));
System.out.println();
}
} // fin de clase
class Hilo extends Thread {
String cadena;
Impresora i;
Hilo(Impresora i, String cadena) {
this.cadena = cadena;
this.i      = i;
}
public void run () {
i.imprimir(cadena);
}
}  // fin de clase
class EjecutaImpresora {
public static void main (String [] args) {
Impresora i = new Impresora();
Hilo h1 = new Hilo (i,"1-Necesitamos Sincronizarnos."+
"2-Necesitamos Sincronizarnos."+
"3-Necesitamos Sincronizarnos."+
"4-Necesitamos Sincronizarnos.");
Hilo h2 = new Hilo (i,"1-Hola Mundo Sincronizado."+
"2-Hola Mundo Sincronizado."+
"3-Hola Mundo Sincronizado.");
Hilo h3 = new Hilo (i,"Adiós Mundo de bucles de sondeo.");
h1.start(); // Se ejecuta el run de la clase Hilo.
h2.start();
h3.start();
}
}  // fin de clase
```

Ejecución EjecutaImpresora sin Synchronized/con Synchronized

Ejecución EjecutaImpresora sin Synchronized

```
java.exe   EjecutaImpresora
```

```
1-Necesitamos Sincronizarnos.2-Necesitamos Sincronizarnos.3-Necesitamos Sincronizarnos.
4-Necesitamos Sincronizarnos.
1-Hola Mundo ASdiinócsr oMnuinzdaod od.e2 -bHloulcal eMsu nddeo  sSoinndceroo.n
izado.3-Hola Mundo Sincronizado.
Process Exit...
```

Ejecución EjecutaImpresora con Synchronized

```
java.exe   EjecutaImpresora
1-Necesitamos Sincronizarnos.2-Necesitamos Sincronizarnos.3-Necesitamos Sincronizarnos.4-
Necesitamos Sincronizarnos.
Adiós Mundo de bucles de sondeo.
1-Hola Mundo Sincronizado.2-Hola Mundo Sincronizado.3-Hola Mundo Sincronizado.
Process Exit...
```

A la espera de que otro hilo termine:

synchronized void <Thread>.join (long milliseconds)
throws InterruptedException

h2.join(10000) // Duerme el hilo.

Número máximo de milisegundos que estará dormido (time out)

Debe ser un Thread (Hilo)

Comunicación entre hilos: Ejemplo

Habrá que sincronizar los hilos de tal forma que el Consumidor espere hasta que termine el Productor y viceversa.

```
class Cola {
    int valor;
    int turno = METER;
    static int METER = 1;
    static int SACAR = 2
```

```
synchronized int get() {
                    while (turno == METER);
                    turno = METER;
                    return valor;
                  }
                synchronized void put(int valor) {
                    while (turno == SACAR);
                    this.valor = valor;
                    turno = SACAR;
                  }
                ) //fin de clase.
```

Gasta muchos recursos

2. Modelo del Productor/ Consumidor

Hay muchas situaciones en las que los subprocesos comparten datos y han de considerar el estado y las actividades de los otros subprocesos.
Una de estas situaciones es el Modelo Productor/Consumidor.

En este modelo, el Productor genera el flujo de datos que son recogidos por el Consumidor. Cuando dos subprocesos comparten un recurso común han de estar sincronizados de algún modo.

El problema se origina cuando el productor va más rápido que el consumidor y genera un segundo dato antes de que el consumidor tenga la oportunidad de recoger el primero. El consumidor se saltará el dato. Del mismo modo, si el consumidor es más rápido que el productor, puede que no tenga datos que recoger, o que recoja varias veces el mismo dato.

Cuando estamos diseñando subprocesos que compiten por recursos limitados hemos de tener cuidado en no caer en dos situaciones extremas, denominadas starvation y deadlock.

starvation : morir de hambre, el subproceso no progresa por falta de recursos.

deadlock: por ejemplo, cuando dos personas están en conflicto y uno está esperando a que el otro tome la iniciativa para resolverlo y viceversa.

El Productor

La clase que describe el Productor, denominada Productor, redefine la función run. Tiene como miembro dato un objeto buffer de la clase Buffer que describiremos más adelante.

Función miembro run

La función miembro run ejecuta un bucle for, cuando se completa el bucle se alcanza el final de run y el subproceso entra en el estado Death (muerto), y detiene su ejecución.

Bucle for

El bucle for se genera una letra al azar y se pone en el objeto buffer llamando a la función poner de la clase Buffer. A continuación se imprime y se hace una pausa por un número determinado de milisegundos llamando a la función sleep y pasándole el tiempo de pausa.

Durante este tiempo el proceso está en estado Not Runnable.

El Consumidor

La clase que describe al Consumidor denominado Consumidor, redefine el método run. La definición de run es similar a la de la clase Productor, salvo que en vez de poner un carácter en el buffer, recoge el carácter guardado en el buffer intermedio llamando a la función recoger.

El tiempo de pausa (argumento de la función sleep) puede ser distinto en la clase Productor que en la clase Consumidor.

Ejemplo

Por ejemplo, para hacer que el productor sea más rápido que el consumidor, se pone un tiempo menor en la primera que en la segunda:

```
public class Consumidor extends Thread {
  private Buffer buffer;
  public Consumidor(Buffer buffer) {
    this.buffer=buffer;
  }
  public void run(){
    char valor;
    for(int i=0; i<10; i++){
       valor=buffer.recoger();
       System.out.println(i+ " Consumidor: "+valor);
       try{
          sleep(100);
       }catch (InterruptedException e) { }
    }
  }
}
```

El buffer

El objeto compartido entre el Productor y el Consumidor está descrito por la clase denominada Buffer. Esta clase tiene dos miembros dato:

1. contenido, guarda un carácter (es el buffer).
2. disponible indica si el buffer está lleno o está vacío, según que esta variable valga true o false. La definición de las funciones poner y recoger es muy simple.

La **función poner** guarda el carácter que se le pasa en su parámetro c en el miembro dato contenido, y pone el miembro disponible en true (el buffer está lleno).

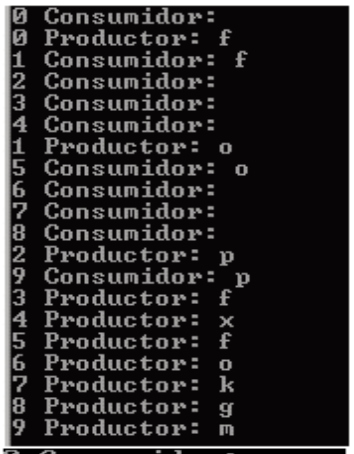
La **función recoger** devuelve el carácter guardado en el miembro contenido, siempre que esté disponible (el buffer lleno, o disponible valga true). En el caso de que no esté disponible (disponible valga false) devuelve un carácter no alfabético: un espacio, un tabulador, lo que desee el usuario.

La aplicación

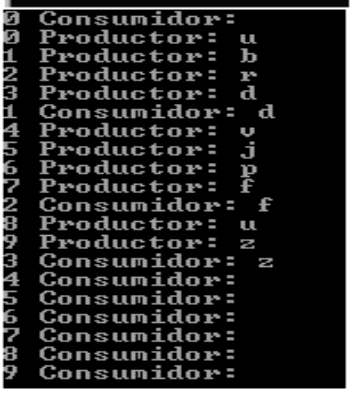
Definimos una clase que describe una aplicación. En la función main, creamos tres objetos: un objeto b de la clase Buffer, un objeto p de la clase Productor y otro objeto c de la clase Consumidor. Al constructor de las clases Productor y Consumidor le pasamos el objeto b compartido de la clase Buffer.

Aplicaciones de la multitarea

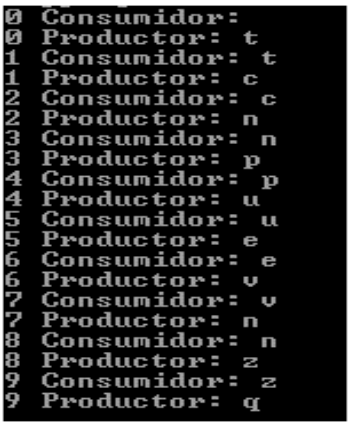
Ponemos en marcha los subprocesos descritos por las clases Productor y Consumidor, mediante la llamada a la función start, de modo que el estado de los subprocesos pasa de New Thread a Runnable. Los subprocesos una vez puestos en marcha mueren de muerte natural, pasando al estado Death después de ejecutar un bucle for de 10 iteraciones en la función run



1. Esta imagen corresponde a la situación en la que el consumidor va más rápido (100 como argumento de sleep) que el productor (400 como argumento de sleep). En la primera iteración (líneas marcadas con 0), el consumidor va al buffer y no encuentra ninguna letra (inicialmente esta vacío), el productor pone en el buffer la letra f. En la segunda iteración (líneas marcadas con 1) el consumidor consume la letra f, y el buffer se queda vacío, el consumidor vuelve a acceder de nuevo al buffer y no encuentra nada, etc.



2. Esta imagen corresponde a la situación en la que el productor va más rápido (100 como argumento de sleep) que el consumidor (400 como argumento de sleep). En la primera iteración (líneas marcadas con 0) el consumidor accede al buffer y lo encuentra vacío, el productor pone en el buffer la letra u. En sucesivas iteraciones el productor pone en el buffer las letras b, a continuación la sustituye por r y luego por d. En la segunda iteración (línea marcada por 1) del consumidor consume esta última letra d,etc.



3. En la tercera imagen, el productor y el consumidor tienen la misma rapidez (100 como argumento de sus funciones sleep). El productor y el consumidor están más coordinados, ya que la letra que pone el productor en el buffer es consumida por él.

   Consumidor. Esta situación es ideal y limitada a unas pocas iteraciones, ya que en general, los dos subprocesos ejecutan tareas distintas durante miles de iteraciones, por lo que no tienen la misma velocidad aún cuando el argumento de la función sleep sea el mismo.

3. Los Applets y los Threads

1. Los Applets y los Threads

Creando animaciones en Java

La animación en Java pasa por construir un patrón de animación y luego pedir a Java que lo dibuje.

Pintando y repintando

El método paint() dibuja el Applet cuando se le llama, cuando cambia la ventana de sitio, cuando la modificamos en tamaño o cuando se superpone encima otra ventana.

Pero podemos llamar a este método más veces para que dibuje modificaciones, pues ésta va a ser la función del método repaint() que nosotros sobrecargaremos (redefiniremos su código) según nos interese.

En nuestro Applet modificaremos el patrón a dibujar y llamaremos a la función repaint(), que primeramente borra la pantalla y después llama al método paint() propiamente para que dibuje en pantalla.

Inicio y parada en la ejecución de un Applet

Necesitamos hacer uso de start() y stop() para dar la señal de comienzo y para parar la ejecución cuando el usuario sale de la página que contiene nuestro Applet.

Uso de Threads

Va a ser necesario utilizar threads para poder trabajar en paralelo con varios programas, en nuestro caso Applets.

Así cada Applet va funcionando en su propio thread sin interferir otros threads.

Si queremos definir algún proceso que se ejecute concurrentemente con el Applet podemos hacer que la clase Applet implemente Runnable y programar en run() las actividades a realizar por los otros hilos.

Escribiendo Applets con threads

Hay cuatro tareas a realizar en el applet relacionadas con la utilización de threads:

a. Cambio de la cabecera de nuestra clase. Se añaden dos palabras que son implements Runnable y que incluyen el interface Runnable necesario para trabajar con threads. Concretamente lo que hace es definir la función run() por defecto.

Mediante las palabras implements Runnable hacemos posible que "otros" puedan llamar a los métodos run() en sus objetos.

b. Incluir una variable instancia para manejar el thread de nuestro Applet. La clase Thread pertenece al paquete java.lang pero no es necesario importarlo. Esta variable se incluye en nuestra clase base (clase que da nombre al Applet).

Thread runner;

c. Añadir un método start() o modificar el ya existente. Un start() típico que se suele utilizar es:

```
public void start() {    // Método start del Applet
if (runner == null) {
runner = new Thread(this);
runner.start();  // Método start del Thread
-
-
-
=
```

Este método crea un nuevo thread y lo pone en marcha.

d. Crear un método run() que contendrá el código de comienzo de nuestro Applet .
Al modificar start() y darle esta apariencia, todo lo que nuestro Applet hacia toda la funcionalidad la hará ahora el método run() que tomará una forma tal que:

```
public void run() {
// código de comienzo de nuestro Applet
// inicializaciones y llamadas a métodos
=
-
```

Método stop()

Finalmente siempre que definamos un método start() para trabajar con threads hemos de definir un método stop() para completar la secuencia de ejecución de nuestro Applet.
Veamos un ejemplo típico de método stop():

```
public void stop() {    //Método stop del Applet
if ( runner != null) {
runner.stop();  //Método stop del Thread
runner = null;
-
=
-
=
```

Este método es para el thread y pone la variable runner que representa nuestro thread a null para que nuestro Applet libere memoria.

Ejemplo completo de un texto que recorre la pantalla del Applet

```
import java.applet.Applet;
import java.awt.*;
public class Texto extends Applet implements Runnable
{
Thread t=new Thread(this);
int x=0;
public void init()
{
setBackground(Color.blue);
setForeground(Color.orange);
}
public void start()
{
t.start();
}
public void stop()
{
t.stop();
}
public void run()
{
while(true)
{
x=x+5;
repaint();
try{
t.sleep(500);
}catch(Exception e){}
}
}
public void paint(Graphics g)
{
g.drawString("Texto en movimiento...",x,20);
}
}
```

Programa Thread

En primer lugar definimos un objeto de la clase Thread que va a representar nuestra tarea (texto que se mueve por la pantalla):

Thread t=new Thread(this);

Después en el método start() del Applet ejecutamos el start() de la tarea, para lograr que la tarea arranque.

t.start();

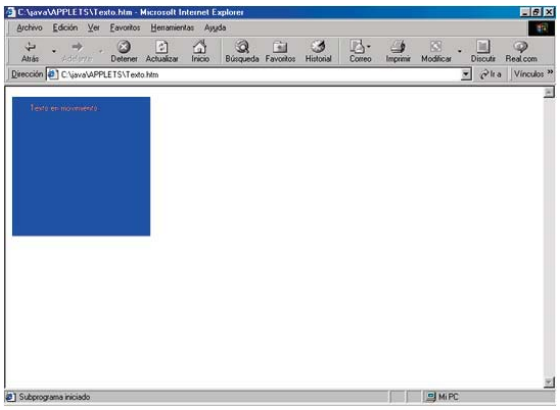
Para forzar a que cada vez que cambiemos de página no siga ejecutándose la tarea dentro de la CPU lo que hacemos es que el stop() de la tarea lo metemos dentro del stop() del Applet, así cuando se ejecute el stop() del Applet forzará a que se ejecute el stop() de la tarea.

Hay que recordar que el stop() del Applet se ejecutaba siempre que cambiábamos a otra página diferente a la que se encontraba el Applet.

El método run() va a representar la CPU y dentro va a haber un bucle while(true) que se detiene siempre que se ejecute el stop() de la tarea.

El método repaint()

El método repaint() lo que hace es que borra la pantalla y llama al método paint() que dibujará la frase pero en una coordenada x mayor, es decir, lo pinta a cada vuelta del bucle un poco más a la derecha.



Código HTML:

```
<html>
  <body>
   <applet code="Texto.class" width=200 height=200>
   </applet>
  </body>
</html>
```

Así se consigue el efecto de movimiento, se pinta algo, se borra y se vuelve a pintar más a la derecha y todo esto con un intervalo de tiempo de por medio.
Cuanto más pequeño sea el intervalo, la sensación será que el texto se desplaza más rápidamente.

Reducción del parpadeo de la animación (Flicker)

El efecto flicker es causado cada vez que un Applet pinta y repinta un patrón (la superficie de ventana del applet).
Decíamos que al llamar a repaint(), ésta implícitamente llamaba a paint(), pero esto no ocurre así exactamente:

- La llamada a repaint() resulta de una llamada al método update().

- El método update() borra la pantalla pintándola del color de fondo y posteriormente llama a paint().

- El método paint() dibuja el nuevo patrón.

Es la llamada a update() lo que produce el parpadeo. Hay dos formas de evitar el flicker:

1. Sobrescribir el método update() para borrar sólo las partes de la pantalla que hayamos cambiado en el nuevo patrón.
2. Sobrescribir ambos: update() y paint().

Flicker: update

Básicamente nuestro nuevo update debe hacer lo mismo, borrar la pantalla y llamar a paint(). Veamos dos formas de hacerlo.

Solución 1

No borrar toda la pantalla. Esto sólo va a servir para algunos Applets.

Se puede, por ejemplo, dibujar lo mismo que había pero en el color del fondo, o dibujar el patrón antiguo otra vez antes del nuevo,...

Un caso sencillo sería redefinir update() como:

```
public void update(Graphics g) {
paint(g);
}
```

Solución 2

Hay cierto movimiento. Vamos a borrar sólo parte del dibujo total, para que no se note tanto el parpadeo.

Redefiniremos update() de forma que sólo borre en una región dada, parámetros de la misma y calcularemos en nuestro método run(). Así una posible implementación de la función update() puede ser:

```
public void update(Graphics g) {
g.clipRect(x,y,w,h);
paint(g);
}
```

Hemos usado el método clipRect(), que borra el área rectangular dada, por la esquina superior izquierda (x,y) la anchura (w) y la altura (h).