Sistemas Concurrentes y Distribuidos.
Práctica 2. Programación de Monitores con Hebras Java.

Dpt. Lenguajes y Sistemas Informáticos
ETSI Informática y de Telecomunicación
Universidad de Granada

Curso 13-14

Índice

Sistemas Concurrentes y Distribuidos.
Práctica 2. Programación de Monitores con Hebras Java.

Indice de la sección

Objetivos

- Conocer cómo construir monitores en Java, tanto usando la API para manejo de hebras Java, como usando un conjunto de clases (el paquete **`monitor`**) que permite programar monitores siguiendo la misma semántica de los monitores de *Hoare*.
- Conocer varios problemas sencillos de sincronización y su solución basada en monitores en el lenguaje Java:
  - Diseñar una solución al problema del *productor-consumidor con buffer acotado* basada en monitores e implementarla con un programa Java multihebra, usando el paquete **`monitor`**.
  - Diseñar una solución al problema de los *fumadores*, visto en la práctica 1, basada en monitores e implementarla con un programa Java multihebra, usando el paquete **`monitor`**.
  - Diseñar e implementar una solución al problema del *barbero durmiente* usando el paquete **`monitor`**.

Indice de la sección

Sección 2
Implementación de monitores nativos de Java

Monitores como Clases en Java

Para construir un monitor en Java, creamos una clase con sus métodos sincronizados. De esta forma solamente una hebra podrá ejecutar en cada momento un método del objeto.
El siguiente ejemplo muestra un monitor que implementa un contador más general que el visto en el seminario:

```
class Contador                          // monitor contador
{ private volatile int actual;
  public Contador( int inicial )
  { actual = inicial ;
  }
  public synchronized void inc()
  { actual++ ;
  }
  public synchronized void dec()
  { actual-- ;
  }
  public synchronized int valor()
  { return actual ;
  }
}
```

Métodos de espera y notificación (1)

- En Java no existe el concepto de variable condición. Podríamos decir que cada monitor en Java tiene una única variable condición anónima.
- Los monitores basados en la biblioteca estándar de Java implementan una versión restringida de la semántica Señalar y Continuar (SC).
- Los mecanismos de espera y notificación se realizan con tres métodos de la clase **`Object`** (los tienen implícitamente todas las clases): **`wait`**, **`notify`** y **`notifyAll`**.
- Estos métodos solamente pueden ser llamados por una hebra cuando ésta posee el cerrojo del objeto, es decir, desde un bloque o un método sincronizado (protegido con **`synchronized`**).

Métodos de espera y notificación (2)

`wait()`

Provoca que la hebra actual se bloquee y sea colocada en una cola de espera asociada al objeto monitor. El cerrojo del objeto es liberado para que otras hebras puedan ejecutar métodos del objeto. Otros cerrojos poseídos por la hebra suspendida son retenidos por esta.

`notify()`

Provoca que, si hay alguna hebra bloqueada en **`wait`**, se escoja una cualquiera de forma arbitraria, y se saque de la cola de **`wait`** pasando esta al estado preparado. La hebra que invocó **`notify`** seguirá ejecutándose dentro del monitor.

`notifyAll()`

Produce el mismo resultado que una llamada a **`notify`** por cada hebra bloqueada en la cola de **`wait`**: todas las hebras bloqueadas pasan al estado preparado.

Métodos de espera y notificación (3)

La hebra señalada deberá adquirir el cerrojo del objeto para poder ejecutarse.

- Esto significará esperar al menos hasta que la hebra que invocó **`notify`** libere el cerrojo, bien por la ejecución de una llamada a **`wait`**, o bien por la salida del monitor.

La hebra señalada no tiene prioridad alguna para ejecutarse en el monitor.

- Podría ocurrir que, antes de que la hebra señalada pueda volver a ejecutarse, otra hebra adquiera el cerrojo del monitor.

Inconvenientes de los monitores nativos de Java

- **Cola de espera única**: como sólo hay una cola de espera por objeto, todas las hebras que esperan a que se den diferentes condiciones deben esperar en el mismo conjunto de espera. Esto permite que una llamada a **`notify()`** despierte a una hebra que espera una condición diferente a la que realmente se notificó, incluso aunque existan hebras esperando la condición que realmente se pretendía notificar. La solución para esta restricción suele consistir en:
  - Sustituir algunas o todas las llamadas a **`notify`** por llamadas a **`notifyAll`**.
  - Poner las llamadas a **`wait`** en un bucle de espera activa:

```
while ( not  condicion_logica_desbloqueo ) { wait() ; }
```

- **No hay reanudación inmediata de hebra señalada**: esto se debe a la semántica SC que se sigue y es la segunda razón del uso de incluir las llamadas a **`wait`** en bucles de espera activa.

Ejemplo: Productor-Consumidor con buffer limitado

Monitor que implementa un buffer acotado para múltiples productores y consumidores:

```
1  class Buffer
2  { private int numSlots = 0, cont = 0;
3    private double[] buffer = null;
4    public Buffer( int p_numSlots )
5    { numSlots = p_numSlots ;
6      buffer = new double[numSlots] ;
7    }
8    public synchronized void depositar( double valor ) throws InterruptedException
9    { while( cont == numSlots ) wait();
10     buffer[cont] = valor; cont++;
11     notifyAll();
12   }
13   public synchronized double extraer() throws InterruptedException
14   { double valor;
15     while( cont == 0 ) wait() ;
16     cont--; valor = buffer[cont] ;
17     notifyAll();
18     return valor;
19   }
20 }
```

Productor-Consumidor: hebras consumidoras

```
47 class Consumidor implements Runnable
48 { private Buffer bb ;
49   int veces;
50   int numC ;
51   Thread thr ;
52   public Consumidor( Buffer pbb, int pveces, int pnumC )
53   { bb    = pbb;
54     veces = pveces;
55     numC  = pnumC ;
56     thr   = new Thread(this,"consumidor "+numC);
57   }
58   public void run()
59   { try
60     { for( int i=0 ; i<veces ; i++ )
61       { double item = bb.extraer ();
62         System.out.println(thr.getName()+", consumiendo "+item);
63       }
64     }
65     catch( Exception e )
66     { System.err.println("Excepcion en main: " + e);
67     }
68   }
69 }
```

Productor-Consumidor: hebras productoras

```
22 class Productor implements Runnable
23 { private Buffer bb ;
24   int veces;
25   int numP ;
26   Thread thr ;
27   public Productor( Buffer pbb, int pveces, int pnumP )
28   { bb    = pbb;
29     veces = pveces;
30     numP  = pnumP ;
31     thr   = new Thread(this,"productor "+numP);
32   }
33   public void run()
34   { try
35     { double item = 100*numP ;
36       for( int i=0 ; i<veces ; i++ )
37       { System.out.println(thr.getName()+", produciendo " + item);
38         bb.depositar( item++ );
39       }
40     }
41     catch( Exception e )
42     { System.err.println("Excepcion en main: " + e);
43     }
44   }
45 }
```

Indice de la sección

Implementación en Java de monitores estilo *Hoare*

- Se ha desarrollado una biblioteca de clases Java (paquete **`monitor`**) que soporta la semántica de monitores estilo *Hoare*.
- Permite definir múltiples colas de condición y un **`signal`** supone la reactivación inmediata del proceso señalado (semántica *Señalar y espera Urgente, SU*).
- La documentación y el código de las clases están disponibles en: http://www.engr.mun.ca/ theo/Misc/monitors/monitors.html
- Para definir una clase monitor específica se debe definir una extensión de la clase **`AbstractMonitor`**.

Exclusión mutua y uso del paquete `monitor`

Para garantizar la exclusión mutua en el acceso a los métodos del monitor, se han de invocar los siguientes métodos de la clase **`AbstractMonitor`**:

- **`enter`**`()`: para entrar al monitor, se invoca al comienzo del cuerpo del método.
- **`leave`**`()`: para abandonar el monitor, se invoca al final del cuerpo (aunque cualquier **`return`** debe ir después).

Uso del paquete

- Es conveniente que el directorio **`monitor`**, conteniendo los archivos `.java` con las clases Java de dicho paquete, se encuentre colgando del mismo directorio donde se trabaje con los programas Java que usen el paquete **`monitor`**.
- En caso contrario, habría que redefinir la variable de entorno `CLASSPATH` incluyendo la ruta de dicho directorio.

Ejemplo: Clase Monitor para contar los días

Permite llevar un control de los días (considerados como grupos de 24 horas) dedicadas por todas las hebras de usuario.

```
3  class Contador_Dias extends AbstractMonitor
4  { private int num_horas = 0, num_dias = 0;
5    public void nueva_hora()
6    { enter() ;
7      num_horas++;
8      if ( num_horas == 24 )
9      { num_dias++;
10       num_horas=0;
11     }
12     leave() ;
13   }
14   public int obtener_dia( )
15   { enter() ;
16     int valor=num_dias;
17     leave() ;
18     return valor;
19   }
20 }
```

Objetos condición del paquete `monitor`

Es posible utilizar varias colas de condición, declarando diversos objetos de una clase denominada **`Condition`**.

- Para crear un objeto condición, se invoca el método **`makeCondition`**`()` de la clase **`AbstractMonitor`**. Ejemplo:

```
Condition puede_leer = makeCondition() ;
```

La clase **`Condition`** proporciona métodos de espera, notificación y para consultar el estado de la cola de condición:

- **`void await`**`()`: tiene la misma semántica que la primitiva **`wait`**`()` de los monitores estilo *Hoare*.
- **`void signal`**`()`: igual que la primitiva **`signal`**`()` de los monitores estilo *Hoare*.
- **`int count`**`()`: devuelve el número de hebras que esperan.
- **`boolean isEmpty`**`()`: indica si la cola de condición está vacía (**`true`**) o no (**`false`**).

Ejemplo: Monitor lectores-escritores (1)

Con prioridad a las lecturas.

```
4  class MonitorLE extends AbstractMonitor
5  {
6    private int num_lectores = 0 ;
7    private boolean escribiendo = false ;
8    private Condition lectura   = makeCondition();
9    private Condition escritura = makeCondition();
10
11   public void inicio_lectura()
12   { enter();
13     if (escribiendo) lectura.await();
14     num_lectores++;
15     lectura.signal();
16     leave();
17   }
18   public void fin_lectura()
19   { enter();
20     num_lectores--;
21     if (num_lectores==0) escritura.signal();
22     leave();
23   }
```

Ejemplo: Monitor lectores-escritores (2)

```
24    public void inicio_escritura()
25    { enter();
26      if (num_lectores>0 || escribiendo) escritura.await();
27      escribiendo=true;
28      leave();
29    }
30    public void fin_escritura() // prio. lect
31    { enter();
32      escribiendo=false;
33      if (lectura.isEmpty()) escritura.signal();
34      else lectura.signal();
35      leave();
36    }
37  } // fin clase monitor "Lect_Esc"
```

Probl. lectores-escritores: Hebra Lectora.

```
52 class Lector implements Runnable
53 {
54   private MonitorLE monitorLE ; // objeto monitor l.e. compartido
55   private int       nveces ; // numero de veces que lee
56   public Thread     thr    ; // objeto hebra encapsulado
57
58   public Lector( MonitorLE p_monitorLE, int p_nveces, String nombre )
59   { monitorLE  = p_monitorLE  ;
60     nveces     = p_nveces ;
61     thr        = new Thread(this,nombre);
62   }
63   public void run()
64   { for( int i = 0 ; i < nveces ; i++ )
65     { System.out.println( thr.getName()+": solicita lectura.");
66       monitorLE.inicio_lectura();
67         System.out.println( thr.getName()+": leyendo.");
68         aux.dormir_max( 1000 ) ;
69       monitorLE.fin_lectura();
70     }
71   }
72 }
```

Probl. lectores-escritores: Hebra Escritora.

```
74  class Escritor implements Runnable
75  {
76    private MonitorLE monitorLE ; // objeto monitor l.e. compartido
77    private int       nveces ; // numero de veces que lee
78    public Thread     thr    ;  // objeto hebra encapsulado
79
80    public Escritor( MonitorLE p_monitorLE, int p_nveces, String nombre )
81    { monitorLE  = p_monitorLE  ;
82      nveces     = p_nveces ;
83      thr        = new Thread(this,nombre);
84    }
85    public void run()
86    { for( int i = 0 ; i < nveces ; i++ )
87      { System.out.println( thr.getName()+": solicita escritura.");
88        monitorLE.inicio_escritura();
89          System.out.println( thr.getName()+": escribiendo.");
90          aux.dormir_max( 1000 );
91        monitorLE.fin_escritura ();
92      }
93    }
94  }
```

Probl. lectores-escritores: clase auxiliar

se ha usado el método (estático) **dormir_max** de la clase **aux**. Este método sirve para bloquear la hebra que lo llama durante un tiempo aleatorio entre 0 y el número máximo de milisegundos que se le pasa como parámetro. Se puede declarar como sigue:

```
39 class aux
40 {
41   static Random genAlea = new Random() ;
42   static void dormir_max( int milisecsMax )
43   { try
44     { Thread.sleep( genAlea.nextInt( milisecsMax ) ) ;
45     }
46     catch( InterruptedException e )
47     { System.err.println("sleep interumpido en 'aux.dormir_max()'");
48     }
49   }
50 }
```

Indice de la sección

Sección 4
Productor-Consumidor con buffer limitado

Ejercicio propuesto

Obtener una versión de la clase **`Buffer`**, que se desarrolló en una sección anterior para múltiples productores y consumidores, usando las clases vistas del paquete **`monitor`**.

Documentación para el portafolio

Los alumnos redactarán un documento donde se responda de forma razonada a cada uno de los siguientes puntos:

1. Describe los cambios que has realizado sobre la clase **`Buffer`** vista anteriormente, indicando qué objetos condición has usado y el propósito de cada uno.
2. Incluye el código fuente completo de la solución adoptada.
3. Incluye un listado de la salida del programa (para 2 productores, 2 consumidores, un buffer de tamaño 3 y 5 iteraciones por hebra.

Indice de la sección

Sección 5
El problema de los fumadores

El problema de los fumadores

En este ejercicio consideraremos de nuevo el mismo problema de los fumadores y el estanquero que ya vimos en la práctica 1:

- Se mantienen exactamente igual todas las condiciones de sincronización entre las distintas hebras involucradas.
- Se escribirá una clase hebra **`Estanquero`** y otra **`Fumador`**. De esta última habrá tres instancias, cada una almacenará el número de ingrediente que necesita (o lo que es equivalente: el número de fumador), que se proporcionará en el constructor.
- La interacción entre los fumadores y el estanquero será resuelta mediante un monitor **`Estanco`** basado en el paquete **`monitor`**.

A continuación se incluyen las plantillas de código que deben usarse para este problema.

Plantilla del monitor `Estanco`

Las sentencias de sincronización se encapsulan en el código del monitor, que tendrá esta declaración:

```
class Estanco extends AbstractMonitor
{  ...
   // invocado por cada fumador, indicando su ingrediente o numero
   public void obtenerIngrediente( int miIngrediente )
   {   ...
   }
   // invocado por el estanquero, indicando el ingrediente que pone
   public void ponerIngrediente( int ingrediente )
   {   ...
   }
   // invocado por el estanquero
   public void esperarRecogidaIngrediente()
   {  ...
   }
}
```

Plantilla de la hebra `Fumador`

Cada instancia de la hebra de fumador guarda su número de fumador (el número del ingrediente que necesita):

```
class Fumador implements Runnable
{
  int miIngrediente;
  public Thread thr ;
  ...
  public Fumador( int p_miIngrediente, ... )
  {  ...
  }
  public void run()
  {
    while ( true )
    { estanco.obtenerIngrediente( miIngrediente );
      aux.dormir_max( 2000 );
    }
  }
}
```

Plantilla de la hebra `Estanquero`

El estanquero continuamente produce ingredientes y espera a que se recojan:

```
class Estanquero implements Runnable
{ public Thread thr ;
  ...
  public void run()
  { int ingrediente ;
    while (true)
    {
      ingrediente = (int) (Math.random () * 3.0); // 0,1 o 2
      estanco.ponerIngrediente( ingrediente );
      estanco.esperarRecogidaIngrediente() ;
    }
  }
}
```

Documentación para el portafolio

Los alumnos redactarán un documento donde se responda de forma razonada a cada uno de los siguientes puntos:

1. Describe qué objetos condición has usado y el propósito de cada uno.
2. Incluye el código fuente completo de la solución adoptada.
3. Incluye un listado de la salida del programa.

Indice de la sección

Sección **6**
El problema del barbero durmiente.

Barbería: Requerimientos de sincronización (1)

El problema del barbero durmiente es representativo de cierto tipo de problemas reales: ilustra perfectamente la relación de tipo cliente-servidor que a menudo aparece entre los procesos.

(1) Una barbería tiene dos puertas y unas cuantas sillas. Los clientes entran por una puerta y salen por otra.

(2) Solamente un cliente o el barbero pueden moverse por el local en cada momento.

(3) El barbero continuamente sirve clientes, uno cada vez. Cuando no hay ningún cliente en la barbería, el barbero está durmiendo en su silla.

(4) Cuando un cliente entra en la barbería y encuentra al barbero durmiendo, lo despierta, se sienta en su silla, y espera a que el barbero termine de pelarlo.

Barbería: Requerimientos de sincronización (2)

(5) Si el barbero está ocupado cuando un nuevo cliente entra, el cliente se sienta en una silla de la sala de espera y espera a que el barbero lo despierte.

(6) Mientras se pela a un cliente, este duerme durante toda la duración del corte.

(7) Cuando el barbero termina de pelar al cliente actual, lo despierta, abre la puerta de salida, y espera a que el cliente salga y cierre la puerta para pasar al siguiente cliente.

(8) A continuación, si hay clientes esperando, el barbero despierta a uno y espera a que se siente en la silla para pelarlo.

(9) Si no hay clientes esperando, el barbero se sienta en su silla y duerme hasta que llegue algún cliente y lo despierte.

Ejercicio propuesto

Escribir un programa Java para el problema del barbero durmiente. Los clientes y el barbero son hebras, y la barbería es un monitor que implementa la interacción entre éstos. Los clientes llaman a **`cortarPelo`** para obtener servicio del barbero, despertándolo o esperando a que termine con el cliente anterior. El barbero llama a **`siguienteCliente`** para esperar la llegada de un nuevo cliente y servirlo. Cuando termina de pelar al cliente actual llama a **`finCliente`**, indicándole que puede salir de la barbería y esperando a que lo haga para pasar al siguiente cliente. Usar las plantillas que se incluyen abajo.

Documentación para el portafolio

Los alumnos redactarán un documento con lo siguiente:

1. Descripción de los objetos condición usados y su propósito.
2. Código fuente completo de la solución.
3. Listado de la salida del programa.

Plantilla del Monitor `Barberia`

```
class Barberia extends AbstractMonitor
{   ...
   // invcado por los clientes para cortarse el pelo
   public void cortarPelo ()
   {   ...
   }
   // invocado por el barbero para esperar (si procede) a un nuevo cliente y sentarlo para el corte
   public void siguienteCliente ()
   {   ...
   }
   // invocado por el barbero para indicar que ha terminado de cortar el pelo
   public void finCliente ()
   {   ...
   }
}
```

Plantilla de las hebras `Cliente` y `Barbero`

```
class Cliente implements Runnable
{ public Thread thr ;
  ...
  public void run ()
  { while (true) {
      barberia.cortarPelo (); // el cliente espera (si procede) y se corta el pelo
      aux.dormir_max( 2000 );   // el cliente está fuera de la barberia un tiempo
    }
  }
}
class Barbero implements Runnable
{ public Thread thr ;
  ...
  public void run ()
  { while (true) {
      barberia.siguienteCliente ();
      aux.dormir_max( 2500 ); // el barbero está cortando el pelo
      barberia.finCliente ();
    }
  }
}
```