Programación multihebra

by Scheme the API man © 2000 (jmrequena@larural.es)

¿Qué es la programación multihebra?

Podemos definir la programación multihebra o multihilo como un estilo de ejecucion en el que se conmuta entre distintas partes del código de un mismo programa durante la ejecución.A cada una de estas partes individuales y atómicas(no divisibles) se les da el nombre de Threads.Los threads-traducidos como hilos,hebras o contextos de ejecución- son espacios de codigo que la JVM ejecutará de forma simultanea en una maquina multiprocesador y de forma conmutada en maquina de un solo procesador,este es,la ejecución de los threads se realizará asignando un determinado quantum o espacio de tiempo a cada uno de los threads.

La motivación inicial que dió lugar a este tipo de programación es la solución de problemas que conllevan la necesidad de estar ejecutando simultaneamente o pseudo-simultaneamente dos tareas al mismo tiempo por el mismo programa,como por ejemplo en el caso de un sistema de control climatico de invernaderos que periodicamente ha de consultar el estado de una serie de sensores repartidos por el complejo.Una primera aproximación a este problema podría ser el realizar un bucle continuo que consultara dentro del mismo proceso el estado de los sensores de forma secuencial,esto es uno tras otro,pero si se trata de sistemas criticos en los que los fallos no se pueden tolerar o bien el tiempo de respuesta en esencial podria darse el caso de que se activara un sensor justo despues de que el bucle lo hubiera comprobado con lo que ese sensor tendria que esperar a que se comprobaran el resto para ser atendido por el programa,con el consiguiente retraso de la respuesta.Otra posibilidad es la ejecución simultanea de multiples procesos,cada uno de los cuales se encargaría de controlar un sensor individualmente pero esta aproximación adolece de un fallo grave,en cualquier contexto la conmutación o cambio entre procesos resulta una tarea extredamente lenta y pesada por lo que el rendimiento del sistema se degradaría.Es en este punto donde cobran sentido los hilos:los hilos son procesos ligeros en el sentido de que cada uno posee su propia pila y conjunto de registros del procesador pero un espacio de direccionamiento compatido,esto es una memoria que puede ser compartida por varios hilos del mismo proceso simultaneamente.La conmutacion entre hilos es tremendamente mas rapida que la conmutacion entre procesos por lo que la alternancia de hilos podría ser una solucion mucho mas eficiente en el problema del sistema de control climatico del invernadero.Inicialmente tendriamos un solo proceso que se encargaría de la gestión inicial de los hilos,de crear y destruir estos hilos y de dar respuesta a las situaciones de alarma,mientras que los hilos lanzados por este proceso atenderían a cada uno de los sensores.

La programación multihilo es una herramienta poderosa y peligrosa.En maquinas monoprocesador,mediante su uso se consigue un mayor rendimiento efectivo pero un menor rendimiento computacional.Esto basicamente quiere decir que el sistema esta ocioso una menor parte del tiempo pero que el numero de instrucciones utiles ejecutadas por unidad de

tiempo desciende ya que al propio proceso de los hilos hay que sumar el tiempo de conmutacion entre ellos.

¿Donde puedo realizar programación multihilo en Java?

En la actualidad casi todos los sistemas operativos proporcian servicios para la programación multihilo pero el tratamiento por parte de la JVM es algo distinto a como lo realizan estos sistemas.Todas las JVM de todas las plataforma proporcionan hilos de ejecución aunque el sistema operatico subyacente no los proporcione,en este caso el JVM simula ese comportamiento.Sin embargo incluso en sistemas que si admiten este tipo de programación hasta la versión del JDK 1.2 los hilos de la JVM no se correspondían a hilos nativos del sistema(excepto en su versión para Solaris) con lo cual el rendimiento del programe en general se veia gravemente dañado.Sin embargo a partir de esta versión la JVM para entornos Windos proporciona servicios de hilos nativos(esto es proporcionados por la api Win32 y no simulados por el sistema) y a partir de la versión 1.3 realiza el mismo tratamiento para entornos Windows.A parte de estos hilos nativos la JVM en todo caso sigue proporcionando hilos simulados por ella misma(no nativos) conocidos como hilos verdes(**green threads**) que garantizan el mismo comportamiento en todas las plataformas en las que se ejecuten en cuanto a la cuestión de la planificación(que hilo toma el control del procesador y en qué momento) pero de un rendimiento sensiblemente mas bajo que los hilos nativos.La decision de usar uno u otro es cuestión de diseño y se escapa al alcance de este tutorial.

Threads en java

En Java un hilo es una clase que desciende de la clase java.lang.Thread o bien que extiende a la interfaz Runnable(util en los casos de que la clase ya forme parte de una jerarquia de clases esto es que ya derive de otra clase).La forma de construccion del hilo(derivación o implementación) es independiente de su utilización,una vez creados se usan de la misma forma sea cual sea el metodo de construcción utilizado.En Java un hilo puede encontrarse en cualquiera de los siguiente estados:

nace	El hilo se declarado pero todavia no se ha dado la orden de puesta en ejecución.(metodo start()).
listo	El hilo esta preparado para entrar en ejecución pero el planificador aun no ha decidido su puesta en marcha
ejecutandose	El hilo esta ejecutandose en la CPU.
dormido	El hilo se ha detenido durante un instante de tiempo definido mediante la utilización del metodo sleep().
bloqueado	El hilo está pendiente de una operación de I/O y no volverá al estado listo hasta que esta termine
suspendido	El hilo ha detenido temporalmente su ejecucion mediante la utilización del metodo suspend() y no la reaundará hasta que se llame a resume().Deprecated,1.2.No debe usarse
esperando	El hilo ha detenido su ejecución debido a una condición externa o interna mediante la llamada a wait() y no reanudará esta hasta la llamada al metodo notify() o notifyAll().
muerto	El hilo ha terminado su ejecución bien porque termino

	de realizar su trabajo o bien porque se llamó al metodo stop().Deprecated 1.2.No debe usarse.

A continuación se puede observar el diagrama de estados explicado previamente:

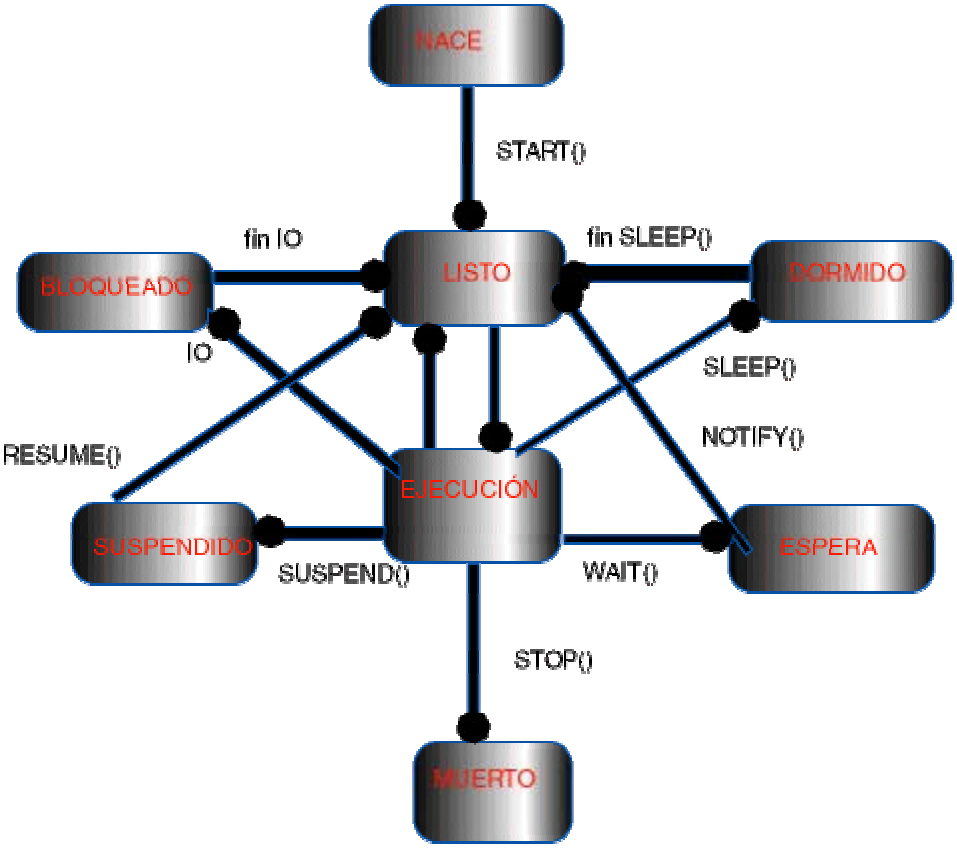

Figura 1.1 Diagrama de estados de un Thread

De todos los metodos que se pueden observar los que no deben utlizarse por estar desautorizados son stop(), suspend() y resume().Su funcionalidad se ha sustituido por otros metodo que se describirán más adelante en este tutorial.

Como habiamos dicho un hilo puede crearse extendiendo a la clase Thread o bien implementando a la interfaz Runnable.En ambos casos la clase debe proporcionar una definicion del metodo run() que contiene el código a ejecutarse una vez que el hilo entre en estado de ejecución.Veamos un ejemplo de los dos metodos:

```
//Clase HiloThread
public class HiloThread extends Thread
{
   public void run()
   {
           while(true)
       {

                System.out.println("Hola mundo,soy el hilo HiloTread");
           }
   }

}

//Clase HiloRunnable

public class HiloRunnable implements Runnable
```

```
{

public void run()
   {
           while(true)
       {

                System.out.println("Hola mundo,soy el hilo HiloRunnable");
           }
   }
}
```

Como se observa ambos metodos son esencialmente muy parecidos en su implementación,pero diferirán algo en su construcción.Bien,ya tengo un par de hilos que simplemente y de forma indefinida escriben en la consola un mensaje de su propia ejecución.¿Como los pongo en marcha y como interactúo con ellos?.Para ello definiremos una nueva clase:

```
//Clase Test



public class Test

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

    //Creamos un hilo del primer tipo
        //y lo ponemos en marcha


    HiloThread ht = new HiloTread();
    ht.start();


        //Creamos un hilo del segundo tipo
        //y lo ponemos en marcha


        Thread hr = new Thread(new HiloRunnable());
        hr.start();

  }
}
```

Como se puede observar ambos hilos se construyen de manera distinta,mientras que el que desciende de Thread se crea como un objeto más el que implementa Runnable,se crea y se pasa al constructor de la clase generica Thread.La puesta en marcha se realiza en ambos casos de forma identica.Se llama al metodo start() y comienza la ejecucion del hilo,mientras no se llame a este metodo el hilo se encontrará en el estado nacido y una vez llamado el hilo entrará en la cola de hilos del planificador hasta el momento en el que este decida su ejecución(recordemos en todo momento hay una seria de hilos compitiendo por el procesador).

La clase Thread proporciona los siguientes constructores para la utilización de hilos:

`Thread()`

`Thread(Runnable target)`

```
Thread(Runnable target, String name)

Thread(String name)

Thread(ThreadGroup group, Runnable target)

Thread(ThreadGroup group, Runnable target, String name)

Thread(ThreadGroup group, String name)
```

De todos estos constructores puede haber dos parametros confusos.Uno es String name que se usa simplemente para dar un nombre el thread de manera que luego pueda referenciarse de forma externa a la referencia que contiene el objeto thread,de esta manera mediante los metodos *String getName() y void setName(String)*.El otro parametro es un objeto del tipo ThreadGroup lo cual merece una explicación mas detallada.

Todo hilo que se este ejecutando en una JVM sin excepciones pertenece a un ThreadGroup,si no se señala ninguno el thread pertenecerá al grupo de hilos principal o grupo "main".La división de hilos en grupos permite una gestión y un tratamiento homogeneo a todos los hilos del grupo como si se tratara de uno solo,pudiendo cambiar su prioridad,detenerlos,interrumpirlos,enumerarlos,obtener información de su estado...etc etc con tan solo una orden.La forma de crear hilos que pertenezcan a un mismo grupo es la siguiente.

```
ThreadGroup group = new ThreadGroup("MiGrupo");

Thread h1 = new Thread(group,new HiloRunnable());
Thread h2 = new Thread(group,new HiloRunnable());
Thread h3 = new Thread(group,new HiloRunnable());
```

group.list(); //muestra información de todos los hilos del grupo

Planificación de hilos

Un tema fundamental dentro de la programación multihilo es la planificación de los hilos.Este concepto se refiere a la politica a seguir de qué hilo toma el control del procesador y de cuando.Obviamente en el caso de que un hilo este bloqueado esperando una operacion de IO este hilo debería dejar el control del procesador y que este control lo tomara otro hilo que si pudiera hacer uso del tiempo de CPU.¿Pero que pasa si hay mas de un hilo esperando?.¿A cual de ellos le otrogamos el control del procesador?.Para determinar esta cuestión cada hilo posee su propia prioridad,un hilo de prioridad mas alta que se encuentre en el estado listo entrará antes en el estado de ejecución que otro de menor prioridad.Cada hilo hereda la prioridad del hilo que lo ha creado,de main si no se crea fuera del metodo run() de otro hilo,habiendo 10 niveles enteros de prioridad desde 1 a 10,siendo 1 la menor prioridad y 10 la mayor prioridad.

h1.setPriority(10); //Le concede la mayor prioridad

h2.setPriority(1); //Le concede la menor prioridad

Tambien existen constantes definidas para la asignación de prioridad estas son :

MIN_PRIORITY=1

NORM_PRIORITY=5

MAX_PRIORITY=10

Bien,¿pero que ocurre en el caso de que haya varios hilos listos en el mismo instante con la misma prioridad?.En est punto debemos hacer la distincion entre hilos nativos e hilos verdes.En un conjunto de hilos nativos la planificación la realizará el sistema operativo de acuerdo a su propia Api(por ejemplo en el caso de NT un MLFQ-multilevel feedback queues de 16+16 grupos divididos en tiempo real y de aplicación) mientras que si se trata de hilos verdes la planificación la realizará la JVM independientemente de en que sistema se este ejecutando mediante un Round Robin de tiempo compartido(la teoria de planificación no la trataremos en este tutorial mas que someramente,se remite al lector a la extensa bibliografía al respecto).

Por tanto nuestras posibilidades de interactuar en la planificación de los hilos se reduce a asignarles una prioridad.A est respecto,y entrando en cuestiones de diseño,citaremos el modelo de Seehein en el que la aplicación está dividida en al menos 3 hilos,uno de logica de negocio,otro de interfaza de usuario y otro de acceso a datos.En este modelo,debería concederse a la interfaz de usuaro la más alta prioridad ya que se encontrará la mayor parte del tiempo inactva esperando una accion por parte del usuario y queremos una respuesta rapida con el fin de proporcionar sensación de viveza de la aplicación.

En una JVM hay dos tipos de hilos los hilos demonio y los que no lo son.Un demonio es un proceso(en este caso hilo) diseñado para dar servicio a otros y que se ejecuta en el background.Podemos determinar o señalar a un hilo como demonio mediante los metodos *boolean isDaemon()* y *void setDaemon(boolean)*.Hay que señalar que cuando en la JVM solo se ejecuten hilos que sean demonios esta se dentendrá y acabará la ejecución del programa,dado que al ser hilos de servicio y no haber otros hilos a los que prestar ese servicio se considera que no queda más que hacer.

Señalaremos por último a este respecto el metodo *yield()(*traducida como ceder el paso).Este metodo pasa al hilo que lo llama al estado de ejecución a listo,permitiendo que el resto de los hilos compitan por el uso de CPU.Es importante en hilos de una alta prioridad y que podrían bloquear el sistema al no dejar al resto de hilos ejecutarse nunca.

Concurrencia y sincronización

La programación concurrente puede dar lugar a muchos errores debido a la utilización de recursos compartidos que pueden ser alterados.Las secciones de codigo potencialmente peligrosas de provocar estos errores se conocen como secciones criticas.

En general los programas concurrentes deben ser correctos totalmente.Este concepto es la suma de dos conceptos distintos la corrección parcial(o safety) esto es que no entrará nunca en ningún mal estado y la correccion temporal(o liveness) esto es que terminará en un algun momento de tiempo finito.En estos programas por tanto hay que evitar que varios hilos entren en una sección critica(exclusión mutua o mutex) y que los programas se bloqueen(deadlock).Ademas los programas que no terminan nunca(programas reactivos) deben cumplir la ausencia de inanición(starvation) esto es que tarde o temprano todos los hilos alcanzen el control del procesador y ninguno quede indefinidamente en la lista de hilos listos y tambien deben cumplir la propiedad de equitatividad(fairwess) esto es repartir el tiempo de la forma mas justa entre todos los hilos.

Un monitor impide que varios hilos accedan al mismo recurso compartido a la vez.Cada monitor incluye un protocolo de entrada y un protocolo de salida.En Java los monitores se consiguen mediante el uso de la palabra reservada *synchronized* bien como instrucción de bloque o bien como modificador de acceso de metodos.Este segundo caso se declararía como sigue:

public synchronized void metodo()

Este metodo declarado de tal forma garantiza que solo un hilo de esta clase puede acceder a este metodo a la vez,de tal forma que si un hilo de la misma clase intenta llamar a este metodo mientras otro hilo esta dentro de el,el hilo que realiza la llamada quedará en estado de bloqueado esperando a que el primer hilo libere el monitor,esto es,salga del metodo sincronizado.Sin embargo las seccione criticas,y el uso de mointores define secciones criticas,es muy costoso en tiempo de ejecucución dado que los demas hilos se detendran hasta que el monitor sea liberado.Y tambien es costoso en tiempo de computación ya que como media se ha determinado que un metodo sincronizado se ejecuta 6 veces máslento que uno que no este sincronizado.Por tanto es recomendable sincronizas sola y exclusivamente aquellas partes del codigo que formen la sección critica y nada mas.A este fin responde el delimitador de bloque *synchronized*,cuyo uso es el siguiente:

```
synchronized(objeto)
{

   . . .

}
```

Durante este bloque los el objeto estará sincronizado.

Sin embargo es posible que el uso de monitores de lugar a interbloqueos(situacion en la que varios hilos no pueden continuar porque estan bloqueados esperando la liberación de un recurso por parte de otros hilos del mismo grupo de tal manera que ninguno avanza).Para solucionar este problema se utilizan dos metodos:

wait()	1. Libera el monitor 2. Suspende la ejecución del hilo y van a para a la cola de espera del monitor(no del procesador no confundir)
notify()	1. Despierta el hilo y les dice que vuelvan a mirar la condición que dió lugar al wait. 2. Si esta se sigue cumpliendo entran otra vez en la cola de espera del monitor y si no es así se marca como listo(que no en ejecución).

Tambien se puede usar el metodo notifyAll() que notifica a todos los hilos que se encuentren en espera y no solo al que realiza la llamada.

EJEMPLO:El problema de los lectores-escritores

A continuación diseñaremos una clase que de solución al problema clasico de los lectores-escritores.En esta situacion tenemos un conjunto de lectores que pueden leer de una fuente

común de forma simultanea,pero nunca deberia haber dos escritores o un escritor y varios lectores en el mismo instante:

```
class LectoresEscritores
{
  int esperandoEscribir=0;
  int nLector=0;
  boolean escribieno=false;


  public synchronized void obtenerLector()
  {
     while(escribiendo || esperandoEscribir > 0)
            wait();
     nLector++;

  }


  public synchronized void liberarLector()
  {
     nLector--;


     if(nLector == 0) notify();

  }


  public synchronized void obtenerEscritor()
  {
    esperandoEscribir++;
    while(nLector > 0 || escribiendo)
               wait();


    escribiendo= true;
        esperandoEscribir--;


  }

  public synchronized void liberarEscritor()
  {
       escribiendo = false;
           notifyAll();

  }

}
```

Un ejemplo para ponerlo todo junto

A continuación desarrollaremos un aplicación que de solución al tambien clasico problema del productor-consumidor.Este problema consiste en dos hilos uno que produce elementos y los deposita en un buffer compartido y otro que lee elementos de ese buffer(consume).En todo momento ha de impedirse que el productor escriba en el buffer sin que el consumidor haya procesado ese elemento y que el consumidor lea sin que el productor haya puesto nada en el buffer.

```
//Clase de testeo


public class Test
{
  public static void main(String args[])
  {
    //Declaramos un buffer
     Buffer b = new Buffer();



    //Iniciamos el productor y el consumidor


     new Thread(new Productor(b)).start();
     new Thread(new Consumidor(b)).start();


  }

}

//Clase buffer



class Buffer
{
   private int n;  //El buffer de 1 elemento
   private boolean leer = false;


   public synchronized void poner(int i)
   {
     try
     {
        while(leer)
           wait();


     }catch(InterruptedException e) { System.out.println(e.getMessage()); }


       n=i;
       System.out.println("Se pone"+i);
       leer=true;
       notify();



   }


       public synchronized int quitar()
    {
     try
     {
        while(!leer)
           wait();


     }catch(InterruptedException e) { System.out.println(e.getMessage()); }
```

```
        n=i;
        System.out.println("Se quita"+n);
        leer=false;
        notify();
            return n;
    }


}


//Clase Productor

class Productor implements Runnable
{
  Buffer bu;

  Productor(Buffer b)
  {
     bu=b;

  }

  public void run()
  {
     int i;
     while(true)  bu.poner(i);

  }

}
//Clase Consumidor

class Consumidor implements Runnable
{
  Buffer bu;

  Consumidor(Buffer b)
  {
     bu=b;

  }

  public void run()
  {
     int i;
     while(true)  bu.quitar(i);

  }

}
```

Bibliografía

Titulo:Java Unleashed 1.2
Autor:Jaime Jaworski
Editorial:Prentice Hall

ISBN:84-8322-061-X

Titulo:Mastering Java 2
Autor:Jhon Zukowski
Editorial:Sybex
ISBN:84-415-0948-4

Titulo:Using Java 1.2
Autor:Mike Morgan
Editorial:Prentice Hall
ISBN:0-7897-1627-5

Titulo:Thinking In Java
Autor:Bruce Eckel
Descargable gratuitamente desde www.bruceEckel.com

Tambien se puede consultar mas información en www.javasoft.com.

Para cualquier duda o matización ponerse en contacto con el autor en jmrequena@larural.es.

Editado y distribuido por www.javahispano.com