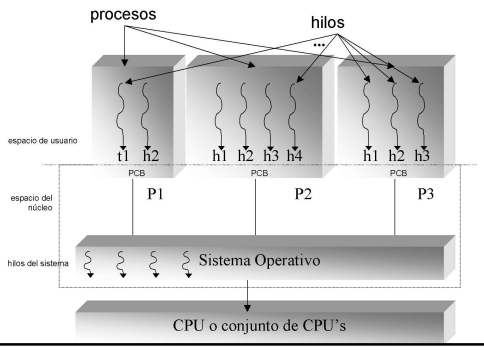
Hilos en Java

Computación Distribuida

Hilos

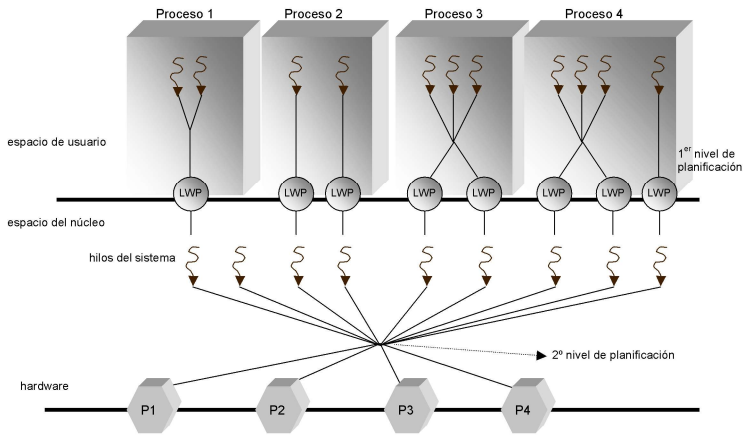
- Un hilo puede definirse como cada secuencia de control dentro de un proceso que ejecuta sus instrucciones de forma independiente.



Hilos vs. Procesos

- Procesos → Entidades Pesadas
  - Llamadas al sistema para acceder a información ubicada en el núcleo
  - Cambio de contextos costosos
- Hilos → Entidades Ligeras
  - Su estructura está almacenada en el espacio de usuario
  - Comparten la información del proceso
  - Cambio de contextos menos costosos

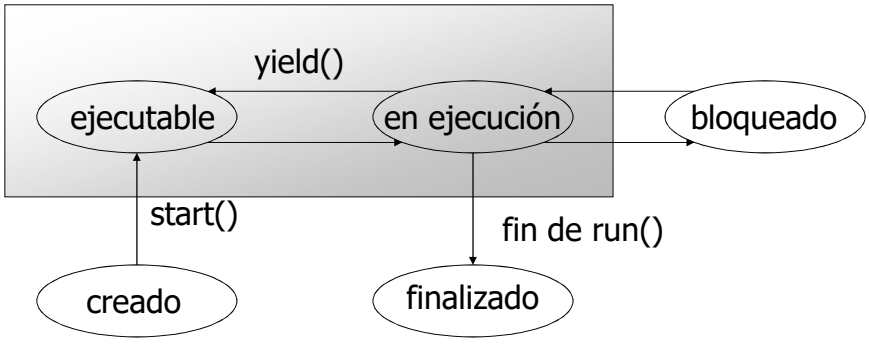
Hilos: Planificación



Programación Multihilos en Java

- Java es un lenguaje interpretado que se ejecuta en la JVM
- La JVM crea un hilo principal en el que se ejecuta el método `main()`
- A partir de este punto el programa puede crear tantos hilos como necesite=> estructura jerárquica
- La JVM se puede considerar un entorno de ejecución en el que pueden estar activos varios hilos

Estados de un hilo en java



Estados de un hilo en java

- **Creado**
  - Hilo creado y preparado para la ejecución
  - `start()` prepara la hilo para su ejecución en la JVM
- **Ejecutable**
  - El hilo está listo para su ejecución en cuanto la JVM lo decida
- **En Ejecución**
  - Se está ejecutando hasta que su tiempo de "CPU" concluya
  - Pasará a finalizado cuando finalice su método **`run()`**
  - Podrá ceder el paso a otros hilo con igual prioridad mediante la ejecución del método **`yield()`**
  - Puede quedar bloqueado mediante la ejecución de una operación de E/S, **`join()`**,**`sleep()`** O **`notify()`**

Estados de un hilo en java

- **Bloqueado**
  - El hilo está inactivo
  - Puede salir de este estado:
    - Acaba el hilo sobre el que se ha realizado el **`join()`**
    - Termina de la operación de E/S que estaba realizando
    - Se despierta de un **`sleep()`**
    - Se ha realizado un **`notify()`** o **`notifyAll();`**
- **Finalizado**
  - Ha finalizado al ejecución del método **`run();`**
  - El recolector de basura libera los recursos ocupados por el hilo

Creación de un Hilo en Java

- La primero forma de crear un hilo en java es crear una clase que herede de la clase **`Thread`**:

```
public class Mihilo extends Thread {

// Variables locales
       public MiHilo(String str){
           super(str);
       // código de inicialización
       }

       public void run(){
       //Código que ejecutará el hilo
       }
}
```

Creación de un Hilo en Java

- Creación del objeto hilo

```
public class PruebaMihilo{

   public static void main(String[] args){
       // Creamos el objeto
       Thread NuevoHilo = new Mihilo("Hilo 1");

       //Iniciamos la ejecución del hilo
       NuevoHilo.start();
}
```

Ejemplo 1

```
public class MiHilo extends Thread {
// Variables locales
  String escribir;
  int veces;

// Constructor
  public MiHilo(String name,String imp,int veces) {
    super(name);
    escribir=imp;
    this.veces=veces;
  }
// Se reescribe el método run
  public void run(){
   for(int i=1;i<veces;i++)
    System.out.println("Thread "+getName()+" imprime "+escribir +
   " i= " + i);
  }
}
```

Ejemplo 1

```
import java.io.*;
public class PruebaMiHilo {
  public static void main(String[] args){
// Se crean los objetos de la clase MiHilo
    Thread Hilo1=new MiHilo("Hilo 1","Hola",100);
    Thread Hilo2=new MiHilo("Hilo 2","Mundo",100);

// Se ejecutan los hilos
    Hilo1.start();
    Hilo2.start();
  }
}
```

Ejemplo 1

```
Thread Hilo 1 imprime Hola i= 1
Thread Hilo 1 imprime Hola i= 2
Thread Hilo 1 imprime Hola i= 3
Thread Hilo 1 imprime Hola i= 4
Thread Hilo 1 imprime Hola i= 5
Thread Hilo 1 imprime Hola i= 6
Thread Hilo 1 imprime Hola i= 7
Thread Hilo 2 imprime Mundo i= 1
Thread Hilo 1 imprime Hola i= 8
Thread Hilo 2 imprime Mundo i= 2
Thread Hilo 1 imprime Hola i= 9
Thread Hilo 2 imprime Mundo i= 3
Thread Hilo 1 imprime Hola i= 10
Thread Hilo 2 imprime Mundo i= 4
Thread Hilo 1 imprime Hola i= 11
Thread Hilo 2 imprime Mundo i= 5
Thread Hilo 2 imprime Mundo i= 6
Thread Hilo 2 imprime Mundo i= 7
Thread Hilo 2 imprime Mundo i= 8
Thread Hilo 2 imprime Mundo i= 9......
```

Ejemplo 2

```
import java.io.*;
public class PruebaDeHilos2 {
  public static void main(String[] args){
// Se crean los objetos de la clase MiHilo
    Thread[] ArrayDeHilos=new Thread[10];
    for(int i=0;i<=9;i++)
     ArrayDeHilos[i]=new MiHilo("Hilo " +(i+1) ,"",10);

// Se ejecutan los hilos
    for(int i=0;i<=9;i++)
     ArrayDeHilos[i].start();
  }
}
```

Ejemplo 2

```
Thread Hilo 1 imprime  i= 1
Thread Hilo 1 imprime  i= 2
Thread Hilo 1 imprime  i= 3
Thread Hilo 1 imprime  i= 4
Thread Hilo 1 imprime  i= 5
Thread Hilo 1 imprime  i= 6
Thread Hilo 1 imprime  i= 7
Thread Hilo 2 imprime  i= 1
Thread Hilo 3 imprime  i= 1
Thread Hilo 4 imprime  i= 1
Thread Hilo 5 imprime  i= 1
Thread Hilo 6 imprime  i= 1
Thread Hilo 7 imprime  i= 1
Thread Hilo 8 imprime  i= 1
Thread Hilo 9 imprime  i= 1
Thread Hilo 10 imprime  i= 1
Thread Hilo 1 imprime  i= 8
Thread Hilo 2 imprime  i= 2....
```

Creación de un Hilo en Java (2)

- Otra forma de crear hilos es implementando la interfaz runnable:

```
public interface Runnable{
    public void run(){}
}
```

```
public class MiHilo extends OtraClase implements Runnable {
//Variables locales
       public MiHilo(String name){
         super(name);
       }
       public void run(){
       // Código
       }
}
```

Creación de un Hilo en Java (2)

- Crear el objeto y ejecutarlo:

```
public class PruebaMiHilo {
       public static void mainString[] args){
       // Creación del objeto
       MiHilo NuevoHilo = new MiHilo(name);
       Thread NuevoThread = new Thread(MiHilo);
       //Se lanza el hilo
       NuevoThread.start();
       }
}
```

Creación de un Hilo en Java (2)

- Crear el objeto y ejecutarlo en una sola línea

```
public class PruebaMiHilo {
       public static void mainString[] args){
       // Creación del objeto y ejecución
       new Thread(new MiHilo()).start();
       }
}
```

Acceso a la Sección Crítica

- Para gestionar el acceso exclusivo a la sección crítica, la clase `Object` de java proporciona una estructura de tipo cerrojo (lock)
- Los cerrojos admiten dos tipos de operaciones cerrar y abrir.
- Cuando un cerrojo está cerrado cualquier otro intento por cerrarlo quedará suspendido hasta que éste sea abierto por el hilo que lo cierre

cerrar(cerrojo)
  Sección Crítica
abrir(cerrojo)

Acceso a la Sección Crítica

- En java las operaciones de cerrar y abrir un cerrojo se gestionan a través de la construcción `synchronized`
- La primera forma de utilizarlo sería mediante un **bloque sincronizado**:

```
Synchronized(Objeto_Dueño_del_
Cerrojo){

        Sección crítica

}
```

Acceso a la Sección Crítica

```
public class MiClase

\\ Variables locales a la clase

    public MiClase(argumentos){
        //constructor
    }

    public void UnMetodo(argumentos){


        Synchronized(this){

            Sección crítica

        }
     }

}
```

Acceso a la Sección Crítica

- El hecho de que un bloque sincronizado exija el objeto que lo posee nos permite definir distintas secciones críticas en una misma clase

```
public class MiClase{
   private Object Cerrojo1= new Object();
   private Object Cerrojo2= new Object();
....
   public int UnMétodo(argumentos){
         synchronized(Cerrojo1){
           Sección Crítica 1
         }
   }
   public void OtroMétodo(argumentos){
         synchronized(Cerrojo2){
           Sección Crítica 2
         }
    }
}
```

Acceso a la Sección Crítica

- La otro forma de utilizar la construcción `synchronized` es la de definir un método sincronizado

```
public class MiClase{

   public synchronized int UnMétodo(argumentos){
         ......
   }


   public synchronized void OtroMétodo(argumentos){
         ......
   }
}
```

Clases sincronizadas ≈ Monitores

- El problema del productor consumidor

```
public class BufferCircular {
  int elementos; // número de elementos en el buffer.
  int pin,pout;  // punteros de inserción y extracción.
  Object[] Buffer;


// Constructor
  public BufferCircular(int tamaño) {
    Buffer = new Object[tamaño];
    elementos = 0;
    pin = 0;
    pout = 0;
  }
```

Clases sincronizadas ≈ Monitores

```
// Insertar un objeto en el buffer
  public synchronized void insertar(Object item){
    Buffer[pin] = item;
    pin = (pin +1) % Buffer.length;
    elementos++;
  }

// Extraer un object del buffer
  public synchronized Object extraer(){
    Object item = Buffer[pout];
    pout = (pout + 1) % Buffer.length;
    elementos--;
    return item;
  }
}
```

Clases sincronizadas ≈ Monitores

```
public class Productor extends Thread{
  BufferCircular Buffer_; // Buffer circular en el que
                          //  se van a insertar los elementos.
  static int Num_Productores = 0; // Número de productores activos

  public Productor(BufferCircular Buffer) {
    Num_Productores++;
    super("Productor" + Num_Productores); // Se crea un Thread con nombre
                                          // "ProductorN"
    Buffer_ = Buffer;
  }

  public void run(){
    while (true) {
      String item = new String("Item"+i); // Producir un elemento
      System.out.println("El productor produce " + item);
      Buffer_.insertar(item);       // Insertar el elemento en el Buffer
    }
  }
}
```

Clases sincronizadas ≈ Monitores

```
public class Consumidor extends Thread{
  BufferCircular Buffer_; // Buffer circular en el que
                          //  se van a insertar los elementos.
  static int Num_Consumidores = 0; // Número de productores activos

  public Consumidor(BufferCircular Buffer) {
    Num_Consumidores++;
    super("Consumidores" + Num_Consumidores);// Se crea un Thread con nombre
                                             // "ConsumidorN"
    Buffer_ = Buffer;
  }

  public void run(){
    while (true) {
        // Extraer un elemento del Buffer
      String item = (String) Buffer_.extraer();
      // Consumir el elemento
      System.out.println("El Consumidor consume el "+ item);
    }
  }
}
```

Clases sincronizadas ≈ Monitores

```
import java.io.*;
public class PruebaProdCon {
  public static void main(String Args[]){
    // Se crea un BufferCircular de tamaño 5
    BufferCircular Buffer = new BufferCircular(5);

    // Se construyen los hilos Productor y Consumidor
    Thread Prod = new Productor(Buffer);
    Thread Cons = new Consumidor(Buffer);

    // Comienza la ejecución de los hilos
    Prod.start();
    Cons.start();

    // Esperamos a que los hilos terminen
    try{
      Prod.join();
      Cons.join();
    }catch(InterruptedException e){}
  }
}
```

Clases sincronizadas ≈ Monitores

- Los `join()` al final se necesitan para hacer que el hilo que ejecuta el método `main()` espere a que los productores y consumidores terminen antes de finalizar el mismo
- El método `join()` lanza una excepción de tipo `InterruptedException` en el caso de que el hilo por el que se espera se interrumpido mediante el método `interrupted()`
- En el ejemplo anterior no hemos tenido en cuenta las condiciones de sincronización

Condiciones de Sincronización: `wait()`

- El método `wait()`:
  - Debe ser llamado dentro de un bloque o método `synchronized`
  - Si el hilo que ejecuta el `wait()` ha sido interrumpido retorna automáticamente lanzando una excepción de tipo `InterruptedException`
  - La JVM pone el hilo que ejecuta el `wait()` en una lista de espera asociada al objeto que contiene la llamada
  - Se abre el cerrojo asociado al objeto
  - Cuando se retorna se vuelve a cerrar el cerrojo
  - Otras formas:
    - `wait(long milisegundos)`
    - `wait(long milisegundos,int nanosegundos)`

Condiciones de Sincronización: notify()

- El método notify():
  - Activará a un hilo que esté esperando en la lista de espera
  - No se sabe que hilo se activará, pero el que se active debe de esperar hasta que pueda cerrar otra vez el cerrojo
  - En nuestro caso no hay confusión, sólo pueden estar esperando o el consumidor o el productor
  - Si existen varios productores o varios consumidores
    - notifyAll() que despierta a todos los procesos de la lista de espera
    - Al no saber cual entrará en ejecución se tiene que volver a comprobar la condición de sincronización

Condiciones de Sincronización

- En la clase BufferCircular hay que modificar:

```
// Insertar un objeto en el buffer
  public synchronized void insertar(Object item){
    try{
        while(elementos == Buffer.length)
            wait();
    } catch(InterruptedException e){}
    Buffer[pin] = item;
    pin = (pin +1) % Buffer.length;
    elementos++;
    notifyAll();
  }
```

Condiciones de Sincronización

```
// Extraer un objeto del buffer
  public synchronized Object extraer(){
  try{
        while(elementos == 0)
            wait();
    } catch(InterruptedException e){}
    Object item = Buffer[pout];
    pout = (pout + 1) % Buffer.length;
    elementos--;
    return item;
    notifyAll();
  }
```

Condiciones de Sincronización

- Una forma de solucionar parte de los problemas consiste en definir colas de espera distintas:

```
public class BufferCircular {
  int elementos; // número de elementos en el buffer.
  int pin,pout;  // punteros de inserción y extracción.
  Object[] Buffer;
  Objetct EsperaPro = new Object();
  Objetct EsperaCon = new Object();


// Constructor
  public BufferCircular(int tamaño) {
    Buffer = new Object[tamaño];
    elementos = 0;
    pin = 0;
    pout = 0;
  }
```

Condiciones de Sincronización

```
// Insertar un objeto en el buffer
  public synchronized void insertar(Object item){
    try{
        while(elementos == Buffer.length)
            EsperaPro.wait();
    } catch(InterruptedException e){}
    Buffer[pin] = item;
    pin = (pin +1) % Buffer.length;
    elementos++;
    EsperaPro.notifyAll();
  }
```

Condiciones de Sincronización

```
// Extraer un objeto del buffer
  public synchronized Object extraer(){
  try{
        while(elementos == 0)
            EsperaCon.wait();
    } catch(InterruptedException e){}
    Object item = Buffer[pout];
    pout = (pout + 1) % Buffer.length;
    elementos--;
    return item;
    EsperaCon.notifyAll();
  }
```

Métodos de la Clase `Thread`

- Ya hemos comentado: `run()`, `start()`, `wait()`, `notify()`, `notifyAll()`
- `static void sleep(long mili)` y `static void sleep(log mili,int nano)`: hace que el hilo actual se suspenda por el tiempo definido
- `void join(long mili)` y `void join(log mili,int nano)`
- `void setName(String name)` y `String getName()`
- `Thread(String name)` y `Thread(Runnable objeto, String name)`
- `static Thread currentThread()`: devuelve el hilo que está siendo ejecutado en ese momento. Puede ser llamado a través de la clase `Thread`: `Thread.currentThread()`
- `static int enumerate(Thread arrayhilo[])`
- `static int activeCount()`