Programación Concurrente con Hilos

Concurrencia

Los usuarios de computadoras toma por hecho que su sistema puede realizar más de una tarea a la vez, asumen que pueden seguir trabajando en un procesador de texto mientras otras aplicaciones descargan archivos, manejan la cola de impresión y el stream de audio. Incluso se suele esperar que una sola aplicación pueda hacer más de una cosa a la vez, por ejemplo, un procesador de texto debería estar listo para responder a eventos de mouse y teclado sin importar lo ocupado que este modificando el texto o actualizando la pantalla.

Un software que sea capaz de realizar estas cosas es conocido como **software concurrente**.

Hilos y Procesos

En la programación concurrente hay dos unidades básicas de ejecución: hilos y procesos.

Procesos

Los procesos tienen un entorno de ejecución autónomo. Un proceso generalmente tiene un conjunto completo y privado de recursos; en particular, cada procesos tiene su propio espacio de memoria.

Para facilitar la comunicación entre procesos, la mayoría de sistemas operativos admite el uso de recursos de comunicación entre procesos (IPC) como pipes o sockets. Los IPC permiten además la comunicación de procesos entre diferentes sistemas.

- **Hilos**

Los hilos también son llamados **procesos ligeros**. Tanto los procesos como los hilos proporcionan un entorno de ejecución, pero crear un nuevo hilo requiere menos recursos que crear un nuevo proceso.

Los hilos existen dentro de un proceso (cada proceso tiene al menos un hilo). Los hilos comparten los recursos del proceso, incluyendo memoria y los archivos abiertos. Esto permite una comunicación eficiente, pero potencialmente problemática.

Desde el punto de vista de un programador de aplicaciones, empezamos con un único hilo conocido como el **main thread**. Este hilo tiene la capacidad de crear hilos adicionales.

Objetos Thread

Cada hilo en java se encuentra asociado con una instancia de la clase Thread. Hay 2 formas de definir e iniciar un hilo.

1. Proporcionar un objeto **Runnable**. La interfaz Runnable define un único método, **run**, el cual debe contener el código a ejecutarse en el hilo.

```
public class HelloRunnable implements Runnable {

    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        (new Thread(new HelloRunnable())).start();
    }

}
```

2. Crear una subclase de **Thread**. La clase Thread implementa por sí mismo la interfaz Runnable, auque su método run no hace nada.

```
public class HelloThread extends Thread {

    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        (new HelloThread()).start();
    }

}
```

Observaciones:

- Ambas formas requieren invocar el método **Thread.start()** para iniciar el nuevo hilo.
- Notemos que al usar el método de la interfaz Runnable es necesario crear una instancia de la clase Thread entregando un objeto Runnable creado como argumento.
- El uso de Runnable es más general pues podemos crear una subclase de cualquier clase que no sea Thread.
- Hacer uso de la segunda forma nos facilita el trabajo en aplicaciones simples, pero nos limita a tener que usar una clase descendiente de la clase Thread.

Thread.currentThread( )

Thread.currentThread( ) es un método estático que retorna una referencia al objeto Thread ejecutándose actualmente.

```
public class ThreadDemo implements Runnable {
    ThreadDemo() {
        // main thread
        Thread currThread = Thread.currentThread();
        Thread t = new Thread(this, "Admin Thread");
        System.out.println("current thread = " + currThread);
        System.out.println("thread created = " + t);
        t.start();
    }
    public void run() {
        System.out.println("This is run() method");
    }
    public static void main(String args[]) {
        new ThreadDemo();
    }
}
```

Thread.sleep( )

Thread.sleep( ) provoca que el hilo actual suspenda su ejecución por un periodo de tiempo especificado.

```
public class SleepMessages {
    public static void main(String args[])
        throws InterruptedException {
        String importantInfo[] = {
            "Mares eat oats",
            "Does eat oats",
            "Little lambs eat ivy"        };
        for (int i = 0; i < importantInfo.length; i++) {
            //Pause for 4 seconds
            Thread.sleep(4000);
            System.out.println(importantInfo[i]);
        }
    }
}
```

- No se garantizan que los tiempos de descanso sean precisos debido a que están limitados por las facilidades proporcionadas por el sistema operativo subyacente.

- El periodo de descanso puede ser detenido por interrupciones, no se puede asumir que invocar sleep suspenda el hilo exactamente por el periodo de tiempo indicado.

- En el ejemplo anterior se declara que main pueda arrojar (throw) una **InterruptedException**. Esta excepción es lanzada cuando algún otro hilo interrumpe el hilo actual mientras sleep está activo.

Interrupciones

- Una **interrupción** es una indicación a un hilo de que debe detenerse y hacer algo más.
- Un hilo envía una interrupción invocando el método **interrupt( )** sobre el objeto Thread para que el hilo sea interrumpido. Para que el mecanismo de interrupción funcione correctamente, el hilo a interrumpir debe ser capaz de manejar su propia interrupción.
- Es decisión del programador el qué hace exactamente un hilo para responder ante una interrupción, pero es bastante común que el hilo termine.

Manejando Interrupciones

La forma de manejar las interrupciones depende de qué este haciendo el hilo en el momento. Si el hilo invoca métodos que arrojen InterruptedException con frecuencia, se suele simplemente retornar del método run luego de atrapar una excepción.

```
for (int i = 0; i < importantInfo.length; i++) {
    // Pause for 4 seconds
    try {
        Thread.sleep(4000);
    } catch (InterruptedException e) {
        // We've been interrupted: no more messages.
        return;
    }
    // Print a message
    System.out.println(importantInfo[i]);
}
```

Thread.interrupted( )

Si un hilo se ejecuta sin utilizar métodos que arrojen interrupciones, se puede hacer invocar periódicamente **Thread.interrupted( )**, el cual retorna true si se ha recibido una interrupción.

```
for (int i = 0; i < inputs.length; i++) {
    heavyCrunch(inputs[i]);
    if (Thread.interrupted()) {
        // We've been interrupted: no more crunching.
        return;
    }
}
```

En aplicaciones más complejas, tiene más sentido arrojar un InterruptedException al detectar una interrupción.

```
if (Thread.interrupted()) {
    throw new InterruptedException();
}
```

La Bandera Interrupt Status

- El mecanismo de interrupción es implementado haciendo uso de una bandera interna conocida como **Interrupt Status**.
- Invocar Thread.interrupt( ) establece esta bandera.
- Cuando un hilo verifica por una interrupción haciendo uso del método estático Thread.interrupted( ), la bandera es limpiada.
- El método no estático **isInterrupted( )**, usado por un hilo para verificar el estado de otro, no modifica la bandera.
- Por convención, cualquier método que termine lanzando una InterruptedException limpia el Interrupt Status al hacerlo.

Join

El método **join( )** permite a un hilo esperar por la terminación de otro. Si *t* es un objeto Thread cuyo hilo se encuentra actualmente en ejecución,

```
t.join();
```

ocasiona que el hilo actual pause su ejecución hasta que el hilo de *t* termine.

- El uso adecuado de join( ) permite al programador especificar un periodo de espera.

- Al igual que sleep( ), join( ) responde a una interrupción saliendo con un Interrupted Exception.

Prioridad de Hilos

Cara hilo cuenta con una prioridad representada por un número entre 1 y 10. Por lo general los hilos se programan acorde a su prioridad, sin embargo, ello no es garantizado debido a que depende de las especificaciones del JVM y cómo decida programar los hilos.

Se cuentan con 3 constantes en la clase Thread:

- MIN_PRIORITY : 1
- NORM_PRIORITY : 5
- MAX_PRIORITY : 10

La prioridad por defecto de un hilo es 5 (NORM_PRIORITY).

Para establecer la prioridad de un hilo u obtener la prioridad actual, se puede hacer uso de los métodos **setPriority( )** y **getPriority( )** respectivamente como se muestra a continuación:

```
class TestMultiPriority1 extends Thread{
   public void run(){
      System.out.println("running thread name is:“  + Thread.currentThread().getName());
      System.out.println("running thread priority is:“  + Thread.currentThread().getPriority());
    }
   public static void main(String args[]){
       TestMultiPriority1 m1=new TestMultiPriority1();
       TestMultiPriority1 m2=new TestMultiPriority1();
       m1.setPriority(Thread.MIN_PRIORITY);
       m2.setPriority(Thread.MAX_PRIORITY);
       m1.start();
       m2.start();
   }
}
```

Daemon Thread

- Un **Daemon Thread** en Java es un hilo de baja prioridad que proporciona servicios a los hilos de usuario.
- Cuando todos los hilos de usuario terminan, el JVM termina los daemon thread automáticamente.
- Un ejemplo de daemon thread es el garbage collector.

MÉTODOS:

- public void setDaemon(boolean status) : marca el hilo sobre el que se usa como daemon thread o user thread.
- public boolear isDaemon( ) : verifica si el hilo es un daemon thread.

Ejemplo de Daemon Thread en Java

```
public class TestDaemonThread1 extends Thread{
    public void run(){
        if(Thread.currentThread().isDaemon()){ //checking for daemon thread
            System.out.println("daemon thread work");
        }
        else{
            System.out.println("user thread work");
        }
    }
    public static void main(String[] args){
        TestDaemonThread1 t1=new TestDaemonThread1(); //creating thread
        TestDaemonThread1 t2=new TestDaemonThread1();
        TestDaemonThread1 t3=new TestDaemonThread1();
        t1.setDaemon(true); //now t1 is daemon thread
        t1.start(); //starting threads
        t2.start();
        t3.start();
    }
}
```

Si se quiere establecer un user thread como daemon, este no debe haberse iniciado o de lo contrario arrojará un **IllegalThreadStateException**.

```
class TestDaemonThread2 extends Thread{
    public void run(){
        System.out.println("Name: "+Thread.currentThread().getName());
        System.out.println("Daemon: "+Thread.currentThread().isDaemon());
    }

    public static void main(String[] args){
        TestDaemonThread2 t1=new TestDaemonThread2();
        TestDaemonThread2 t2=new TestDaemonThread2();
        t1.start();
        t1.setDaemon(true); //will throw exception here
        t2.start();
    }
}
```