Unidad Threads (Hilos)

Indice

1. Introducción

Los programas realizan actividades o tareas, y para ello pueden seguir uno o más flujos de ejecución. Dependiendo del número de flujos de ejecución, podemos hablar de dos tipos de programas:

- Programa de flujo único: Es aquel que realiza las tareas una a continuación de la otra, de manera secuencial, lo que significa que cada una de ellas debe concluir por completo, antes de que pueda iniciarse la siguiente.

- Programa de flujo múltiple: Es aquel que coloca las tareas a realizar en diferentes flujos de ejecución, de manera que cada uno de ellos se inicia y termina por separado, pudiéndose ejecutar éstos de manera simultánea o concurrente.

La **programación multihilo** consiste en desarrollar programas o aplicaciones de flujo múltiple. Cada uno de esos flujos de ejecución es un hilo.

2. Conceptos sobre hilos

Un hilo, denominado también subproceso, es un flujo de control secuencial independiente dentro de un proceso y está asociado con una secuencia de instrucciones, un conjunto de registros y una pila.

Podemos hacer las siguientes observaciones:

• Un hilo no puede existir independientemente de un proceso.
• Un hilo no puede ejecutarse por si solo.
• Dentro de cada proceso puede haber varios hilos ejecutándose.
• Un único hilo es similar a un programa secuencial; por si mismo no nos ofrece nada nuevo.
• Es la habilidad de ejecutar varios hilos dentro de un proceso lo que ofrece algo nuevo y útil, ya que cada uno de estos hilos puede ejecutar actividades diferentes al mismo tiempo.

Hilo: los hilos, o threads, son la unidad básica de utilización de la CPU, y más concretamente de un core del procesador. Así un thread se puede definir como la secuencia de código que está en ejecución, pero dentro del contexto de un proceso.

Hilo vs Proceso: hasta ahora hemos visto que el sistema operativo gestiona procesos, asignándoles la memoria y recursos que necesiten para su ejecución. En este sentido, el sistema operativo planifica únicamente procesos.

Los hilos se ejecutan dentro del contexto de un proceso, por lo que dependen de un proceso para ejecutarse. Mientras que los procesos son independientes y tienen espacios de memoria deferentes,

dentro de un mismo proceso pueden coexistir varios hilos ejecutándose que compartirán la memoria de dicho proceso. Esto sirve para que el mismo programa en ejecución (proceso) pueda realizar diferentes tareas (hilos) al mismo tiempo. Un proceso siempre tendrá, por lo menos, un hilo de ejecución que es el encargado de la ejecución del proceso.

2.1. Recursos compartidos por los hilos

Un hilo lleva asociados los siguientes elementos:

◦ Un identificador único.
◦ Un contador de programa propio.
◦ Un conjunto de registros.
◦ Una pila (variables locales).

Por otra parte, un hilo puede compartir con otros hilos del mismo proceso los siguientes recursos:

◦ Código.
◦ Datos (como variables globales).
◦ Otros recursos del sistema operativo, como los ficheros abiertos y las señales.

El hecho de que los hilos compartan recursos (por ejemplo, pudiendo acceder a las mismas variables) implica que sea necesario utilizar esquemas de bloqueo y sincronización, lo que puede hacer más difícil el desarrollo de los programas y así como su depuración.

2.2. Ventajas y uso de hilos

Como consecuencia de compartir el espacio de memoria, los hilos aportan las siguientes ventajas sobre los procesos:

◦ Se consumen menos recursos en el lanzamiento y la ejecución de un hilo que en el lanzamiento y ejecución de un proceso.
◦ Se tarda menos tiempo en crear y terminar un hilo que un proceso.
◦ La conmutación entre hilos del mismo proceso es bastante más rápida que entre procesos.

Es por esas razones, por lo que a los hilos se les denomina también procesos ligeros.

Se aconseja utilizar hilos en una aplicación cuando:
◦ La aplicación maneja entradas de varios dispositivos de comunicación.
◦ La aplicación debe poder realizar diferentes tareas a la vez.
◦ Interesa diferenciar tareas con una prioridad variada. Por ejemplo, una prioridad alta para manejar tareas de tiempo crítico y una prioridad baja para otras tareas.
◦ La aplicación se va a ejecutar en un entorno multiprocesador.

3. Multihilo en Java. Librerías y clases

3.1. Utilidades de concurrencia del paquete java.util.concurrent

Concretamente estas utilidades están dentro de los siguientes paquetes:

◦ java.util.concurrent: En este paquete están definidos los siguientes elementos:
- ▪ Clases de sincronización: Semaphore, CountDownLatch, CyclicBarrier y Exchanger.
- ▪ Interfaces para separar la lógica de la ejecución: Executor, ExecutorService, Callable y Future.
- ▪ Interfaces para gestionar colas de hilos: BlockingQueque, LinkedBlokingQueque, nArrayBlockingQueque, SynchronousQueque, PriorityBlockingQueque y DelayQueque.

◦ java.util.concurrent.atomic: Incluye un conjunto de clases para ser usadas como variables atómicas en aplicaciones multihilo y con diferentes tipos de dato, por ejemplo AtomicInteger y AtomicLong.

◦ java.util.concurrent.locks: Define una serie de clases como uso alternativo a la cláusula sinchronized. En este paquete se encuentran algunas interfaces como por ejemplo Lock, ReadWriteLock.

4. Creación de hilos

En Java, un hilo se representa mediante una instancia de la clase java.lang.Thread.
Este objeto Thread se emplea para iniciar, detener o cancelar la ejecución del hilo de ejecución.

Los hilos o threads se pueden implementar o definir de dos formas:

- Extendiendo la clase Thread.
- Mediante la interfaz Runnable.

En ambos casos, se debe proporcionar una definición del método run(), ya que este método es el que contiene el código que ejecutará el hilo, es decir, su comportamiento.

Extender la clase Thread es el procedimiento más sencillo, pero no siempre es posible. Si la clase ya hereda de alguna otra clase padre, no será posible heredar también de la clase Thread (recuerda que Java no permite la herencia múltiple), por lo que habrá que recurrir al otro procedimiento.

Implementar Runnable siempre es posible, es el procedimiento más general y también el más flexible.

Para saber qué hilo se está ejecutando en un momento dado, utilizamos el método currentThread() y obtenemos su nombre invocando al método getName(), ambos de la clase Thread.

4.1. Creación de hilos extendiendo la clase Thread

Para definir y crear un hilo extendiendo la clase Thread, haremos lo siguiente:

◦ Crear una nueva clase que herede de la clase Thread.
◦ Redefinir en la nueva clase el método run() con el código asociado al hilo.
◦ Crear un objeto de la nueva clase Thread. Éste será realmente el hilo.
◦ Una vez creado el hilo, para ponerlo en marcha o iniciarlo invocamos al método start() del objeto thread.

4.2. Creación de hilos mediante la interfaz Runnable

Para definir y crear hilos implementando la interfaz Runnable seguiremos los siguientes pasos:
◦ Declarar una nueva clase que implemente a Runnable.
◦ Redefinir en la nueva clase el método run() con el código asociado al hilo.
◦ Crear un objeto de la nueva clase.
◦ Crear un objeto de la clase thread pasando como argumento al constructor el objeto cuya clase tiene el método run().
◦ Una vez creado el hilo, para ponerlo en marcha o iniciarlo invocamos al método start() del objeto thread.

5. Estados de un hilo

Los diferentes estados en los que se puede encontrar un hilo son los siguientes:

• Nuevo: Se ha creado un nuevo hilo, pero aún no está disponible para su ejecución.

• Ejecutable: El hilo está preparado para ejecutarse. Puede estar Ejecutándose, siempre y cuando se le haya asignado tiempo de procesamiento, o bien que no esté ejecutándose en un instante determinado en beneficio de otro hilo, en cuyo caso estará Preparado.

• No Ejecutable o Detenido: El hilo podría estar ejecutándose, pero hay alguna actividad interna al propio hilo que se lo impide, como por ejemplo una espera producida por una operación de Entrada/Salida (E/S). Si un hilo está en estado "No Ejecutable", no tiene oportunidad de que se le asigne tiempo de procesamiento.

• Muerto o Finalizado: El hilo ha finalizado. La forma natural de que muera un hilo es finalizando su método run().

El método getState() de la clase Thread, permite obtener en cualquier momento el estado en el que se encuentra un hilo. Devuelve por tanto: NEW, RUNNABLE, NO RUNNABLE o TERMINATED.

5.1. Iniciar un hilo

Para que el hilo se pueda ejecutar, debe estar en el estado "Ejecutable" y para conseguir ese estado es necesario iniciar o arrancar el hilo mediante el método start() de la clase Thread().

El método start() realiza las siguientes tareas:

◦ Crea los recursos del sistema necesarios para ejecutar el hilo.
◦ Se encarga de llamar a su método run() y lo ejecuta como un subproceso nuevo e independiente.

Algunas consideraciones importantes que debes tener en cuenta son las siguientes:
◦ Puedes invocar directamente al método run(), por ejemplo poner hilo1.run() y se ejecutará el código asociado a run() dentro del hilo actual (como cualquier otro método), pero no comenzará un nuevo hilo como subproceso independiente.
◦ Una vez que se ha llamado al método start() de un hilo, no puedes volver a realizar otra llamada al mismo método. Si lo haces, obtendrás una excepción IllegalThreadStateException.
◦ El orden en el que inicies los hilos mediante start() no influye en el orden de ejecución delos mismos, lo que pone de manifiesto que el orden de ejecución de los hilos es no- determinístico (no se conoce la secuencia en la que serán ejecutadas la instrucciones del programa).

5.2. Detener temporalmente un hilo

Un hilo pasará al estado "No Ejecutable" o "Detenido" por alguna de estas circunstancias:

◦ El hilo se ha dormido: Se ha invocado al método sleep() de la clase thread, indicando el tiempo que el hilo permanecerá deteniendo. Transcurrido ese tiempo, el hilo se vuelve "Ejecutable", en concreto pasa a "Preparado".
◦ El hilo está esperando: El hilo ha detenido su ejecución mediante la llamada al método wait() y no se reanudará hasta que se produzca una llamada al método notify() o notifyAll() por otro hilo.
◦ El hilo se ha bloqueado: El hilo está pendiente de que finalice una operación de E/S en algún dispositivo, o a la espera de algún otro tipo de recurso.
El método suspend() (actualmente en desuso) también permite detener temporalmente un hilo, y en ese caso se reanudaría mediante el método resume() (también en desuso).

5.3. Finalizar un hilo

La forma natural de que muera o finalice un hilo es cuando termina de ejecutarse su método run(), pasando al estado 'Muerto'. Una vez que el hilo ha muerto, no lo puedes iniciar otra vez con start().
Si en tu programa deseas realizar otra vez el trabajo desempeñado por el hilo, tendrás que:
◦ Crear un nuevo hilo con new().
◦ Iniciar el hilo con start().

Se puede utilizar el método isAlive() de la clase Thread para comprobar si un hilo está vivo o no. Un hilo se considera que está vivo (alive) desde la llamada a su método start() hasta su muerte. IsAlive() devuelve verdadero (true) o falso (false), según que el hilo esté vivo o no.
El método stop() de la clase Thread (actualmente en desuso) también finaliza un hilo, pero es poco seguro.

6. Gestión y planificación de hilos.

La ejecución de hilos se puede realizar mediante:

• Paralelismo: En un sistema con múltiples CPU, cada CPU puede ejecutar un hilo diferente.
• Pseudoparalelismo: Si no es posible el paralelismo, una CPU es responsable de ejecutar múltiples hilos.

La ejecución de múltiples hilos en una sola CPU requiere la planificación de una secuencia de ejecución (sheduling). El planificador de hilos de Java (Sheduler) utiliza un algoritmo de secuenciación de hilos denominado fixed priority scheduling que está basado en un sistema de prioridades relativas, de manera que el algoritmo secuencia la ejecución de hilos en base a la prioridad de cada uno de ellos.

7. Sincronización y comunicación de hilos

¿Cómo conseguimos que los hilos se ejecuten de manera coordinada? Utilizando sincronización y comunicación de hilos:

• Sincronización: Es la capacidad de informar de la situación de un hilo a otro. El objetivo es establecer la secuencialidad correcta del programa.
• Comunicación: Es la capacidad de transmitir información desde un hilo a otro. El objetivo es el intercambio de información entre hilos para operar de forma coordinada.

En Java la sincronización y comunicación de hilos se consigue mediante:
• Monitores: Se crean al marcar bloques de código con la palabra synchronized.
• Semáforos: Podemos implementar nuestros propios semáforos, o bien utilizar la clase Semaphore incluida en el paquete java.util.concurrent.
• Notificaciones: Permiten comunicar hilos mediante los métodos wait(), notify() y notifyAll() de la clase java.lang.Object.

7.1. La clase Semaphore

La clase Semaphore del paquete java.util.concurrent, permite definir un semáforo para controlar el acceso a un recurso compartido.

Para crear y usar un objeto Semaphore haremos lo siguiente:

◦ Indicar al constructor Semaphore (int permisos) el total de permisos que se pueden dar para acceder al mismo tiempo al recurso compartido. Este valor coincide con el número de hilos que pueden acceder a la vez al recurso.
◦ Indicar al semáforo mediante el método acquire(), que queremos acceder al recurso, o bien mediante acquire(int permisosAdquirir) cuántos permisos se quieren consumir al mismo tiempo.
◦ Indicar al semáforo mediante el método release(), que libere el permiso, o bien mediante release(int permisosLiberar), cuantos permisos se quieren liberar al mismo tiempo.

Hay otro constructor Semaphore (int permisos, boolean justo) que mediante el parámetro justo permite garantizar que el primer hilo en invocar adquire () será el primero en adquirir un permiso cuando sea liberado garantizando un orden secuencial de acceso.

¿Desde dónde se deben invocar estos métodos? Esto dependerá del uso de Semaphore.

◦ Si se usa para proteger secciones críticas, la llamada a los métodos acquire() y release() se hará desde el recurso compartido o sección crítica, y el número de permisos pasado al constructor será 1.
◦ Si se usa para comunicar hilos, en este caso un hilo invocará al método acquire() y otro hilo invocará al método release() para así trabajar de manera coordinada. El número de permisos pasado al constructor coincidirá con el número máximo de hilos bloqueados en la cola o lista de espera para adquirir un permiso.

2. Información compartida entre hilos

Las secciones críticas son aquellas secciones de código que no pueden ejecutarse concurrentemente, pues en ellas se encuentran los recursos o información que comparten diferentes hilos, y que por tanto pueden ser problemáticas.

La forma de proteger las secciones críticas es mediante sincronización.
La sincronización se consigue mediante:

◦ Exclusión mutua: Asegurar que un hilo tiene acceso a la sección crítica de forma exclusiva y por un tiempo finito.
◦ Por condición: Asegurar que un hilo no progrese hasta que se cumpla una determinada condición.

En Java, la sincronización para el acceso a recursos compartidos se basa en el concepto de monitor.

3. Monitores. Métodos synchronized

En Java, un monitor es una porción de código protegida por un mutex o lock. Para crear un monitor en Java, hay que marcar un bloque de código con la palabra synchronized.

Añadir synchronized a un método significará que:
◦ Hemos creado un monitor asociado al objeto.
◦ Solo un hilo puede ejecutar el método synchronized de ese objeto a la vez.
◦ Los hilos que necesitan acceder a ese método synchronized permanecerán bloqueados y en espera.
◦ Cuando el hilo finaliza la ejecución del método synchronized, los hilos en espera de poder ejecutarlo se desbloquearán. El planificador Java seleccionará a uno de ellos.

4. Monitores. Segmentos de código synchronized.

Hay casos en los que no se puede, o no interesa sincronizar un método. La forma de resolver esta situación es poner las llamadas a los métodos que se quieren sincronizar dentro de segmentos sincronizados de la siguiente forma:

```
synchronized (objeto){
        //sentencias segmento;
    }.
```

En este caso el funcionamiento es el siguiente:
◦ El objeto que se pasa al segmento, es el objeto donde está el método que se quiere sincronizar.
◦ Dentro del segmento se hará la llamada al método que se quiere sincronizar.

◦ El hilo que entra en el segmento declarado synchronized se hará con el monitor del objeto, si está libre, o se bloqueará en espera de que quede libre. El monitor se libera al salir el hilo del segmento de código synchronized.

◦ Solo un hilo puede ejecutar el segmento synchronized a la vez.

5. Comunicación entre hilos con métodos de java.lang.Object.

La comunicación entre hilos la podemos ver como un mecanismo de auto-sincronización, que consiste en logar que un hilo actúe solo cuando otro ha concluido cierta actividad (y viceversa).

Java soporta comunicación entre hilos mediante los siguientes métodos de la clase java.lang.Object:

◦ wait(): Detiene el hilo (pasa a "no ejecutable"), el cual no se reanudará hasta que otro hilo notifique que ha ocurrido lo esperado.

▪ wait(long tiempo): Como el caso anterior, solo que ahora el hilo también puede reanudarse(pasar a "ejecutable 🗣) si ha concluido el tiempo pasado como parámetro.

◦ notify(): Notifica a uno de los hilos puestos en espera para el mismo objeto, que ya puede continuar.

▪ notifyAll(): Notifica a todos los hilos puestos en espera para el mismo objeto que ya pueden continuar.

6. El problema del interbloqueo (deadlock)

El interbloqueo o bloqueo mutuo (deadlock) consiste en que uno a más hilos, se bloquean o esperan indefinidamente.

¿Cómo se llega a una situación de interbloqueo?

◦ Porque cada hilo espera a que le llegue un aviso de otro hilo que nunca le llega.

◦ Porque todos los hilos, de forma circular, esperan para acceder a un recurso.

Otro problema, menos frecuente, es la inanición (starvation), que consiste en que un hilo es desestimado para su ejecución. Se produce cuando un hilo no puede tener acceso regular a los recursos compartidos y no puede avanzar, quedando bloqueado. Esto puede ocurrir porque el hilo nunca es seleccionado para su procesamiento o bien porque otros hilos que compiten por el mismo recurso se lo impiden.