LPS: Hilos y sincronización

Federico Peinado
www.federicopeinado.es

Depto. de Ingeniería del Software e Inteligencia Artificial
disia.fdi.ucm.es

Facultad de Informática
www.fdi.ucm.es

Universidad Complutense de Madrid
www.ucm.es

Programación paralela

- La ejecución de tareas en paralelo optimiza la utilización de los recursos del sistema
  - Ejemplo: cuando un proceso está esperando la finalización de una operación de E/S, otros procesos pueden aprovechar el procesador del sistema que en ese momento no se usa
- Las máquinas actuales pueden ejecutar varios programas simultáneamente
  - En teoría: sólo tantos como *procesadores (núcleos)* tengan
  - En la práctica: intercalando la ejecución de varios procesos, cambiando muy rápido entre ellos, "parece" como si se ejecutaran simultáneamente (programación concurrente)
- La programación paralela/concurrente es mucho más compleja que la convencional

Procesos e hilos

- Son las unidades básicas de ejecución de la programación concurrente
- Procesos
  - Disponen de su propio espacio de memoria
  - Se pueden comunicar a través de *pipes* y *sockets*
- Hilos (*threads*, también llamados *hebras*)
  - Ejecuciones simultáneas dentro de un mismo proceso (podemos considerarlos como "procesos ligeros")
  - Su espacio de memoria es, por tanto, compartido
- Java pone más énfasis en los hilos que en los procesos
  - La propia máquina virtual de Java es un único proceso con múltiples hilos
  - *Java 6 sí propone nuevos mecanismos de control de procesos, que no vamos a estudiar aquí…*

Hilos en Java

Hilos en Java

- Cuando se inicia un programa en Java, la máquina virtual crea un hilo principal
  - El hilo se encargará de invocar al método *main* de la clase que se comienza a ejecutar
  - El hilo termina cuando se acaba de ejecutar el método *main*
  - Si el hilo principal crea otros hilos, éstos comenzarán su ejecución de forma concurrente
  - Sólo cuando no queda ningún hilo activo, es cuando se termina el programa

Thread

- La clase principal para conseguir concurrencia en Java es la clase Thread
  - Dispone de un método *start()* que ocasiona la ejecución del código que tenga dentro de su método *run()* en un nuevo hilo
- Todos los hilos se ejecutan en la misma máquina virtual (mismo proceso)
  - Por tanto comparten recursos, como la memoria
  - En realidad sólo puede haber un hilo ejecutándose a la vez (se alternan, gracias a la concurrencia)

Creación de hilos

- Esquema general
  - Se implementar el método run, cuyo código define lo que va a hacer el hilo durante su ejecución
  - Se *crear el hilo* y se llama a su método start, que se encarga, entre otras cosas, de llamar a run
- Dos alternativas posibles para la creación
  1. Implementando la interfaz Runnable
  2. Heredando de la clase Thread

Interfaz Runnable

```
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();
    }
}
```

- Lo más recomendable para aplicaciones de tamaño medio-grande
  - Es más flexible y permite combinar este interfaz con lo que heredemos de cualquier otra clase
  - Permite gestión de hilos de alto nivel mediante el API de Java: **`java.util.concurrent`**

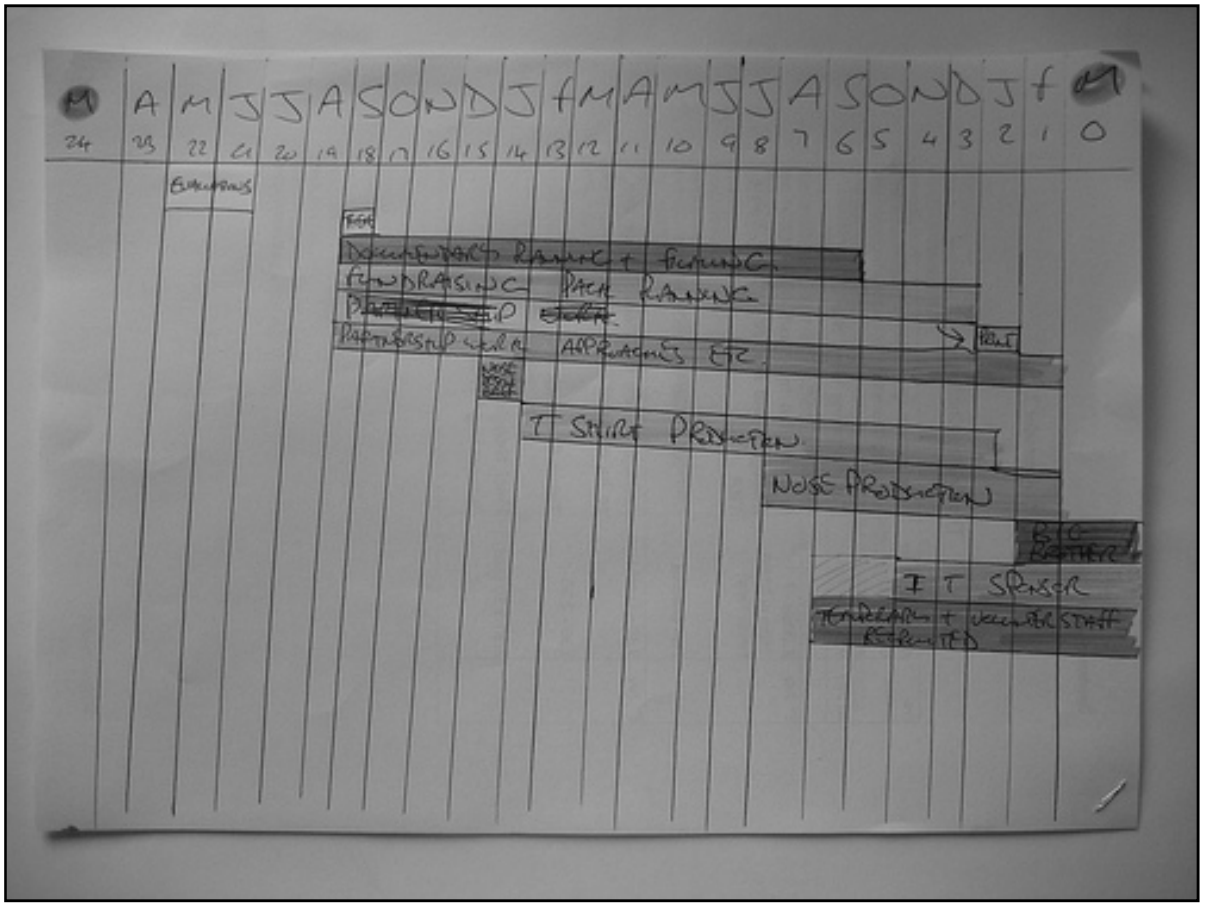
Clase Thread

```
public class HelloThread extends Thread {
   public void run() {
       System.out.println("Hello from a thread!");
   }

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

- Lo más cómodo para aplicaciones simples, al resultar ligeramente más sencillo que lo anterior

Planificación de tareas



Hilos en la MVJ

- El planificador de la máquina virtual de Java (MVJ) decide qué hilo ejecutar en cada momento
  - La especificación no hace explícito el algoritmo a utilizar
- Hay cuestiones dependientes de la implementación
  - La *creación de un hilo* en el S.O. subyacente
  - La *simulación de los hilos* dentro de la MVJ
- Sí es obligatorio que el planificador tenga en cuenta las prioridades de los hilos
  - Si hay varios hilos con la misma prioridad, todos se deben ejecutar en algún momento
  - No es obligatorio que deban ejecutarse hilos de menor prioridad si hay pendientes algunos con mayor prioridad

Ajustando la prioridad

- La clase Thread tiene dos métodos para trabajar con las prioridades
  - **`setPriority(int)`** establece la prioridad del hilo. Puede generar una excepción si:
    - El parámetro es inválido, o
    - El hilo que invoca al método no tiene los permisos necesarios para cambiarla
  - **`getPriority()`** devuelve la prioridad del hilo
- Para los valores de las prioridades, existen tres constantes estáticas en la clase Thread
  - **`MAX_PRIORITY`** (=10): prioridad máxima
  - **`MIN_PRIORITY`** (=1): prioridad mínima
  - **`NORM_PRIORITY`** (=5): prioridad por defecto.

Ejemplo conocido: Swing

- Tanto Swing como AWT crean un hilo nuevo
  - Sirve para procesar los eventos de entrada (teclado y ratón)
- Consecuencias
  - Es problemático leer de teclado desde el hilo principal con Swing lanzado: *dos hilos leen simultáneamente del mismo dispositivo*
  - Cuando hay eventos, el hilo de Swing es quien invoca los métodos de usuario de procesado de eventos
- Para que la interacción con el usuario sea fluida, el hilo de Swing tiene prioridad 6, y los hilos normales 5
  - Cuando hay un evento de entrada, responde inmediatamente
  - Cuando no hay eventos, el hilo está suspendido a la espera de los mismos para no perjudicar a los otros hilos

Problema habitual en Swing

- Si la gestión del evento (hilo de Swing) lleva mucho tiempo, el resto de los eventos no se procesan
  - ¡La ventana parece no reaccionar a las órdenes del usuario!
- Evitar crear métodos largos de gestión de eventos
  - Bloquean el funcionamiento de la GUI hasta que terminan
- Si el procesado de un evento va a durar demasiado, el gestor debería:
  1. Recopilar toda la información de los componentes Swing adecuados
  2. Crear un nuevo hilo que realice la tarea en paralelo

Gestión de hilos

Paradas voluntarias

- **`void sleep(long t)`** duerme el hilo durante *al menos* t milisegundos
  - Cuando transcurran, el hilo estará preparado para ejecutarse
  - El planificador lo lanzará cuando considere oportuno
- **`void sleep(long milis, int nanos)`** versión con más precisión (a nivel de nanosegundos)
  - En la práctica, las implementaciones no tienen tanta precisión… ☹
- Las dos versiones del método **`sleep`** pueden generar la excepción **`InterruptedException`** que hay que capturar

- **`void yield()`** pausa temporalmente el hilo
  - No queda suspendido, sino que sigue estando preparado para la ejecución
  - El planificador se activa y carga otro hilo para ejecutar, que *podría ser el mismo*

Interrupciones exteriores

- ¡No se puede parar la ejecución de un hilo desde fuera!
  - En realidad sí, pero se desaconseja (*métodos obsoletos*)…
- Para parar un hilo de ejecución, tenemos que solicitárselo mediante **`interrupt()`**, y confiar en que nos hará caso
- El hilo debería comprobar periódicamente si lo quieren parar
  - Algunos métodos de la librería de Java generan una excepción si durante su ejecución se les intenta interrumpir (como es el caso de **`sleep()`**)
  - Así se comprueba si nos han mandado una interrupción:
    - **`Thread.interrupted()`**
  - Así se comprueba si otro hilo ha recibido una interrupción:
    - **`isInterrupted()`**

Ejemplo: Hilo obediente 1

```
// Hilo que presenta un mensaje importante cada 4 segundos
public class HiloObediente1 implements Runnable {
  // ...

  public void run() {
    for (int i = 0; i < importantInfo.length; i++) {
        // Paramos 4 segundos
        try {
            Thread.sleep(4000);
        } catch (InterruptedException e) {
            // Nos han pedido terminar. Obedecemos
            return;
        }
        // Escribimos el mensaje
        System.out.println(importantInfo[i]);
     }
  }
  // ...
}
```

Ejemplo: Hilo obediente 2

```
// Hilo que multiplica 10000 matrices de 1000x1000
public class HiloObediente2 extends Runnable {
  // ...

  void run() {
    for (int i = 0; i < 10000; i++) {
      this.multiplicaMatriz(i);
      if (Thread.interrupted())
        // Nos han pedido que acabemos...
        break; // o return;
    }
  }

  // ...
}
```

Otras funciones

- Esperar la ejecución de un hilo **`t`** para continuar la ejecución
  - **`t.join()`**
- Comprobar si un hilo está activo
  - **`t.isAlive()`**

Sincronización y concurrencia

Sincronización

- El hecho de que varios hilos compartan el mismo espacio de memoria puede causar dos problemas
  1. Interferencias
  2. Inconsistencias
- Algunos hilos necesitan *esperar a otros* para disponer de datos importantes o evitar problemas de acceso simultáneo a recursos
  - Java proporciona mecanismos de sincronización de hilos para tratar estos problemas

Métodos sincronizados

- Proporcionan exclusión mutua para acceder a un recurso compartido
- Uso de **`synchronized`** al declarar un método

```
public synchronized void metodo()
```

  - No es posible ejecutar simultáneamente dos métodos sincronizados *del mismo objeto*
- Sincronizando todo, se pierde la ventaja de la concurrencia... pero se puede sincronizar únicamente la parte del código del método que interesa:

```
public void addName(String name) {
  //...
  synchronized ( this ) {
     // Código importante, peligro de concurrencia
  }
  //...
}
```

Sincronización con monitores

- Los métodos sincronizados se implementan utilizando monitores
  - Todas las referencias (*objetos* y *arrays*) disponen de un monitor interno que sólo permite el acceso a un hilo de ejecución a la vez
- Concepto de monitor
  - Estructura de alto nivel para la gestión de concurrencia
  - El monitor sólo permite el acceso a un hilo a la vez
  - Contiene una lista priorizada de hilos en espera para entrar
  - Gestiona dicho acceso
- Podemos crear objetos nuevos para usarlos como monitores

Comunicación entre hilos

- Cuando dos o más hilos son interdependientes, deben decidir entre ellos cuándo esperar y cuándo avanzar
- Sistema de parada y avance
  - Cuando un hilo A depende de que otro B complete una tarea, se para a esperar
  - Cuando el hilo B completa su tarea, B avisa a A para que continúe su ejecución
- Implementación incorrecta: espera "activa"

```
public void esperaCondicion ( ) {
  while ( !condicion ) { //No hacer nada }
  // Seguir ejecutando…
}
```

Instrucciones de parada y avance

- Todos los objetos implementan los siguientes métodos referidos a su monitor interno
  - **`wait()`** Detiene la ejecución del hilo hasta recibir una notificación
  - **`notify()`** Despierta a uno de los hilos detenidos
    - ¡No se puede controlar especificamente a cual!
  - **`notifyAll()`** Despierta a todos los hilos detenidos
    - Todos pasan a estar "en espera" hasta que el planificador les ceda el turno, claro
- La gestión de la parada y el avance está integrada en el monitor
  - Para invocar el método **`wait`** es necesario tener el control del monitor... por tanto, *siempre debe activarse dentro de un método sincronizado*

Esperando a una condición

- No se garantiza que, cuando un hilo despierta, lo hace porque ya se cumple la condición a la que estaba esperando, por lo que la llamada a **`wait`** se debe hacer en un **`while`**, en vez de en un **`if`**
- En el hilo que espera:

```
while(!condicion) {
   try {
      [this.]wait();
   } catch (InterruptedException e) {}
}
```

- En el otro hilo:

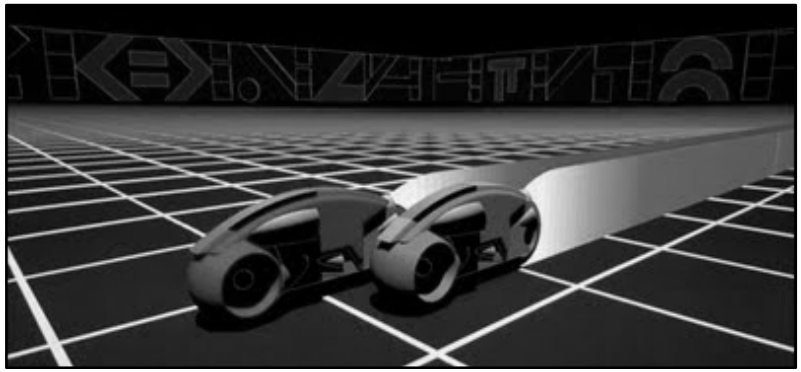
```
t.notifyAll();
```

Problemas de sincronización

- Interbloqueo o punto muerto (*deadlock*)
  - Los hilos no se ejecutan por quedarse a la espera de la ejecución de otros hilos
- Innanición (*starvation*)
  - Un hilo no puede acceder a un recurso compartido porque otros hilos no le permiten el acceso
- Interbloqueo activo (*livelock*)
  - Los hilos no se ejecutan porque se estorban mutuamente, precisamente por intentar ambos evitar un interbloqueo

Uso de hilos en la práctica

- Generalmente se usan un mínimo de dos hilos en las prácticas (con interfaz gráfico)
  - El hilo principal
    - Crea todas las ventanas y después, idealmente, muere porque se termina el *main*
  - El hilo de Swing
    - Gestiona los eventos del usuario (se acaba convirtiendo en el hilo principal, digamos)



- Si se necesitan inteligencias artificiales o entidades que realicen tareas por su cuenta (como sub-servidores para tratar con múltiples clientes), se usan *hilos paralelos* el tiempo que haga falta

Críticas, dudas, sugerencias…

Federico Peinado
www.federicopeinado.es