Programación secuencial vs programación paralela

Hasta ahora hemos hecho programación secuencial, cuando lanzamos cualquier tarea como ordenar un vector, ésta se ejecuta en un núcleo hasta que termina y luego comienza la siguiente tarea.
En programación paralela, podremos lanzar varias tareas a la vez y cada una se ejecutará en un núcleo diferente.
Si no ejecutamos dos cosas a la vez no tendremos nunca concurrencia o paralelismo y por tanto tendremos programación secuencial.
Además **NO** es lo mismo concurrencia que paralelismo (Lo explicaremos más adelante).
Con la concurrencia conseguiremos un incremento de rendimiento muy significativo en nuestro software.
Actualmente el hecho de que muchos ordenadores presentan varios núcleos en sus microprocesadores permite que nuestro software pueda ser mucho más potente.
La concurrencia permite que se puedan hacer tareas en segundo plano y programar tareas asíncronas.
El inconveniente es que los programas serán más complejos y sobretodo se presentarán problemas indeterminación.

En la programación concurrente necesitamos saber lanzar hilos.

Hilos

Un hilo es básicamente una forma que tenemos de planificar una serie de tareas. Anteriormente programabamos una tarea, después otra y luego la siguiente. Con la programación concurrente queremos que se ejecuten dos o más tareas a la vez. Un hilo al ser un planificador de tareas nos permitirá hacer esta ejecución de tareas según nos interese. Ver (*Ejemplo_0000_Crear_Un_Hilo_Run_Y_Thr*) y (*Ejemplo_0000A_Crear_Un_Hilo_Con_Parametros*).

Si nos interesa que el hilo principal (el del Main) se ejecute después que algún otro hilo de los que fueron lanzados por el Main podemos utilizar el método join() de cada hilo. Ver (*Ejemplo_0000AA_Lanzar_Hilos_Desde_Un_Vector*)

Indeterminismo

Si tenemos una variable estática (static int var;…) y la incrementamos por 1000 hilos que acceden concurrentemente a esa variable, veremos que el resultado es indeterminado. Recordad que una variable estática es aquella que la comparten todos los objetos de esa clase. Las variables estáticas son variables de clase y no de cada objeto individual. Ver (*Ejemplo_0000AAA_Indeterminismo*).
La única forma de solucionar esto es mediante el concepto de "**exclusión mutua**", es decir que a esa variable solo pueda acceder un hilo a la vez. Es necesario controlar el acceso a la

"**sección crítica**" que es aquella sección en la que se puede producir indeterminismo en el caso del ejemplo esta sección sería donde incrementamos contador "cont++". Podremos controlar el acceso a la "sección crítica" mediante la "exclusión mutua".
En el ejemplo *Ejemplo_0000AAAA_Una_Posible_Solucion_Indeterminismo* vemos otro problema de indeterminismo y una posible forma de solucionarlo sería en *Ejemplo_0000AAAAA_Solucion_Indetermismo*. En esta solución los dos hilos trabajan a la vez con el mismo elemento, sólo que cada uno accede a una sección distinta del elemento común.
Otro ejemplo similar pero esta vez con una matriz sería el siguiente *Ejemplo_0000AAAAA_Otra_solucion_Indeterminismo_Sinchronize*
Otro ejemplo que además nos muestra cuántos hilos puede ejecutar mi CPU a la vez y que crearía tantos hilos como CPU´s tenga mi procesador de forma que utilice toda la potencia de la máquina destino sería Ejemplo_0000AAAAAA_Otra_solucion_indeterminismo_hilos_dinamicos.

Sin embargo, en las soluciones anteriores controlabamos el indeterminismo simplemente accediendo a secciones distintas de la variable que provocaba el indeterminismo. En la siguiente solución *Ejemplo_0000AB_Primera_Solucion_Indeterminismo_Con_Sincronize*, lo que hacemos es controlar el indeterminismo impidiendo que 2 o más hilos puedan acceder a la vez a la "**sección crítica**" mediante la exclusión mutua de los hilos a dicha sección, esto lo hacemos controlando el acceso mediante **synchronized**. Otro ejemplo distinto que ayuda a entender cómo funciona el **synchronized** además de explicar el patrón de diseño **Singleton** es *Ejemplo_0000B_Patron_Singlenton*, en éste ejemplo el modificador **synchronized** se coloca en la definición del método en lugar de sobre un objeto estático. Esto se hace para impedir que varios hilos ejecuten ese método a la vez.
En el ejemplo *Ejemplo_0000AB_Primera_Solucion_Indeterminismo_Con_Sincronize* no podíamos controlar el orden de ejecución, en el ejercicio *Ejemplo_0000ABB_Otra_Solucion_Indeterminismo_wait_notifyAll* se propone una forma de controlar el orden de ejecución. Hay que tener en cuenta que en éste ejemplo se forman dos colas la formada por el synchronized (obj) y la cola formada por el obj.wait(). El método obj.notifyAll() sólo libera los hilos que se quedaron esperando en el wait(), los hilos que se quedaron esperando en el synchronized forman una cola distinta (leer comentarios del código).
En el ejemplo *Ejemplo_0000ABBB_Interbloqueo_Entre_Hilos* se produce un **"interbloqueo entre Hilos"** ya que el hilo del Main nunca ejecuta la instrucción "println" del final porque al poner el join() al resto de hilos, éste no se puede ejecutar hasta que el resto de hilos no terminen.
En el ejemplo *Ejemplo_0000ABBBB_Interbloqueo_Solucion* se optimiza muchísimo el código para incrementar el contador, además se elimina el problema del interbloqueo.