Programación multiproceso

Índice

1. Ejecutables. Procesos. Servicios.

1.1. Ejecutables

Un ejecutable es un archivo con la estructura necesaria para que el sistema operativo pueda poner en marcha el programa que hay dentro. En Windows, los ejecutables suelen ser archivos con la extensión .EXE.

Se pueden utilizar «desensambladores» para averiguar la secuencia de instrucciones que hay en un EXE. Incluso existen desensambladores en línea como http://onlinedisassembler.com

Sin embargo, Java genera ficheros .JAR o .CLASS. Estos ficheros no son ejecutables sino que son archivos que el intérprete de JAVA (el archivo java.exe) leerá y ejecutará.

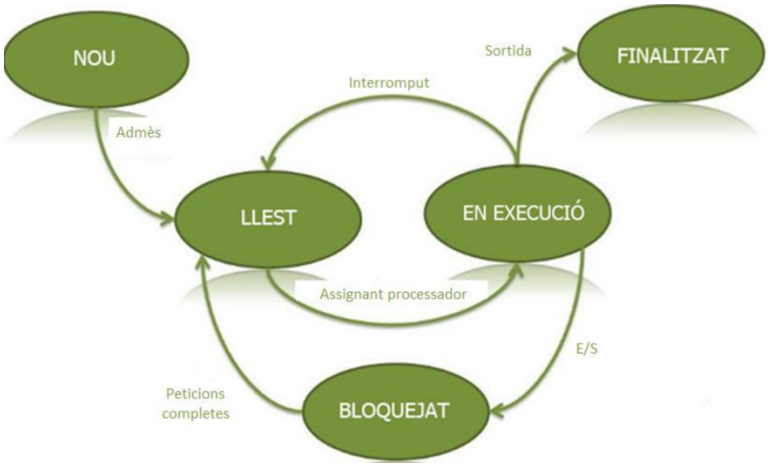
Un archivo .CLASS puede desensamblarse utilizando el comando javap -c <archivo.class> . Cuando se hace así, se obtiene un listado de «instrucciones» que no se corresponden con las instrucciones del microprocesador, sino con «instrucciones virtuales de Java». El intérprete Java (el archivo java.exe) traducirá en el momento del arranque dichas instrucciones virtuales Java a instrucciones reales del microprocesador.

Este último aspecto es el esgrimido por Java para defender que su ejecución puede ser más rápida que la de un EXE, ya que Java puede averiguar en qué microprocesador se está ejecutando y así generar el código más óptimo posible.

Un EXE puede que no contenga las instrucciones de los microprocesadores más modernos. Como todos son compatibles no es un gran problema, sin embargo, puede que no aprovechemos al 100% la capacidad de nuestro micro.

1.2. Procesos

Es un archivo que está en ejecución y bajo el control del sistema operativo. Un proceso puede atravesar diversas etapas en su «ciclo de vida». Los estados en los que puede estar son:



Nuevo: El proceso se acaba de crear pero aún no ha sido admitido en el grupo de procesos ejecutables del sistema operativo. En los sistemas operativos raras veces un proceso queda en el estado de Nuevo sino que pasa directamente a Listo, pero se podría dar el caso en situaciones donde existe sobrecarga de procesos o no existan recursos.

Listo: El proceso no se esta ejecutando pero puede hacerlo en cualquier momento. La CPU o procesador mantiene colas de los procesos ordenadas por diferentes criterios como (prioridad, tiempo de ejecución, afinidad a un procesador, etc). Dado un momento, la CPU solo puede dedicar-lo a un proceso así que el proceso queda en Listo hasta que el llamado "momento" sea dado al proceso en cuestión.

En ejecución: El proceso se está ejecutando en la CPU.

Bloqueado o también llamado "Suspendido" en Linux: Los procesos en este estado no pueden pasar a ejecución, tienen que esperar a que ocurra un evento para pasar a Listo.

Parado: El proceso ha salido del grupo de procesos ejecutables del sistema operativo, liberando así los recursos utilizados, como por ejemplo, la memoria.

Los procesos pasan a bloqueados porque solicitaron una operación de Entrada o Salida (E/S) al núcleo y deben esperar a que acabe. Cuando la operación de E/S finaliza, el proceso es movido a la cola de listos para ejecutar.

1.3. Servicios

Un servicio es un proceso que no muestra ninguna ventana ni gráfico en pantalla porque no está pensado para que el usuario lo maneje directamente.

Habitualmente, un servicio es un programa que atiende a otro programa.

2. Hilos.

Un hilo es un concepto más avanzado que un proceso i para entender bien que es, se tienen que ver las diferencias con un proceso.
Un hilo es considerado una tarea que puede ser ejecutado al mismo tiempo que otra tarea (concurrencia), compitiendo por los recursos.

2.1. Diferencias Proceso - Hilo

Al hablar de procesos cada uno tiene su propio espacio en memoria. Si abrimos 20 procesos cada uno de ellos consume 20x de memoria RAM. Un hilo es un proceso mucho más ligero, en el que el código y los datos se comparten de una forma distinta.

Un proceso no tiene acceso a los datos de otro procesos. Sin embargo un hilo sí accede a los datos de otro hilo (espació, ficheros, etc). Esto complicará algunas cuestiones a la hora de programar.

Los procesos son típicamente independientes y se comunican entre ellos a través de mecanismos de comunicación dados por el sistema. Los hilos, comparten memoria, es decir, acceden a las mismas variables sin la necesidad de mecanismos de comunicación complejos.

2.2. Ejemplos

Por ejemplo, nos dicen de realizar la creación de un programa que haga una serie de cálculos independientes al mismo tiempo: Se podría crear un programa que de forma secuencial realizara todos los cálculos solicitados, los cálculos se realizarían de forma independiente pero uno tras otro, en lugar de eso, si se crea un hilo por cada operación independiente, el programa podrá ejecutar todos los cálculos de forma independiente y al mismo tiempo. (Sin entrar en la discusión de que solo se puede procesar una operación en un momento en la CPU).

2.3. Problemática

Es posible que los hilos requieran de operaciones atómicas para impedir que los datos comunes se cambien o sean leídos al mismo tiempo que son modificados. No tener estas problemáticas en cuenta da lugar a errores graves de programación.

3. Programación concurrente.

La programación concurrente es la parte de la programación que se ocupa de crear programas que pueden tener varios procesos/hilos que colaboran para ejecutar un trabajo y aprovechar al máximo el rendimiento de sistemas multinúcleo. En el caso de la programación concurrente un solo ordenador puede ejecutar varias tareas a la vez (lo que supone que tiene 2 o más núcleos). Ejemplo: Escuchar música, imprimir documentos y visualizar la pantalla del ordenador al mismo tiempo.

En el caso de ejecutar un programa concurrente en un único procesador con un único núcleo se llama multiprogramación.

- En este caso, puede parecer que se ejecutan diversos programas al mismo tiempo pero no es así, solo un proceso puede estar en un momento determinado en ejecución.
- La programación concurrente no mejora el tiempo de ejecución global de los programas.

La programación concurrente tiene ventajas pero no son gratuitas. La compartición de recursos (fundamentalmente memoria) tiene riesgos y puede provocar errores difíciles de detectar y depurar. Debido al carácter naturalmente asíncrono y no determinista de la ejecución de los procesos ya no es posible tratar a los procesos concurrentes como una ejecución secuencial de instrucciones.

3.1. Programación paralela y distribuida.

Dentro de la programación concurrente tenemos la paralela y la distribuida:

- En general se denomina «programación paralela» a la forma de ejecutar programas concurrentes buscando siempre la eficiencia asintótica de programas con comportamiento determinista. Ejemplo: Utilizar los diferentes núcleos de un procesador para ejecutar sentencias de forma coordinada.

- Se denomina «programación distribuida» a la creación de software que se ejecuta en ordenadores distintos y que se comunican a través de una red para compartir los recursos necesarios.
  De está forma un programa puede ejecutar distintos procesos en ordenadores distintos para y mejorar así el rendimiento de la ejecución. Esto tiene un coste, la comunicación para compartir los recursos es más compleja y costosa.

4. Creación de procesos.

En Java es posible crear procesos utilizando algunas clases que el entorno ofrece para esta tarea. En este tema, veremos en profundidad la clase ProcessBuilder.

El ejemplo siguiente muestra como lanzar un proceso de Acrobat Reader:

```
public class LanzadorProcesos {
        public void ejecutar(String ruta){

                ProcessBuilder pb;
                try {
                        pb = new ProcessBuilder(ruta);
                        pb.start();
                } catch (Exception e) {
                        // TODO Auto-generated catch block
                        e.printStackTrace();
                }

        }
        /**
         * @param args
         */
        public static void main(String[] args) {
                String ruta=
                        "C:\\Program Files (x86)\\Adobe\\Reader 11.0
                LanzadorProcesos lp=new LanzadorProcesos();
                lp.ejecutar(ruta);
                System.out.println("Finalizado");
        }

}
```

Supongamos que necesitamos crear un programa que aproveche al máximo el número de CPUs para realizar alguna tarea intensiva. Supongamos que dicha tarea consiste en sumar números.

Enunciado: crear una clase Java que sea capaz de sumar todos los números comprendidos entre dos valores incluyendo ambos valores.

Para resolverlo crearemos una clase Sumador que tenga un método que acepte dos números n1 y n2 y que devuelva la suma de todo el intervalo.

Además, incluiremos un método main que ejecute la operación de suma tomando los números de la línea de comandos (es decir, se pasan como argumentos al main).

El código de dicha clase podría ser algo así:

```
package com.ies;

public class Sumador {
        public int sumar(int n1, int n2){
                int resultado=0;
                for (int i=n1;i<=n2;i++){
                        resultado=resultado+i;
                }
                return resultado;
        }
        public static void main(String[] args){
                Sumador s=new Sumador();
                int n1=Integer.parseInt(args[0]);
                int n2=Integer.parseInt(args[1]);
                int resultado=s.sumar(n1, n2);
                System.out.println(resultado);
        }
}
```

Este código será suficiente para poder leer dos números pasados por parámetro y mostrar la suma.

Ahora, el siguiente paso es poder ejecutar este código mediante un comando de terminal y poder pasar así los dos parámetros indicados. En cada SO será diferente, en el mío por ejemplo es:

java -jar /Users/joanbarcelo/NetBeansProjects/P1/dist/P1.jar

Cuando compiláis (Build) en el programa IDE, en la salida por consola gràfica os debería aparecer el comando que ejecuta el programa para arrancar el código que habéis diseñado.

compile:

Created dir: /Users/ joanbarcelo /NetBeansProjects/P1/dist

Copying 1 file to /Users/ joanbarcelo /NetBeansProjects/P1/build

Nothing to copy.

Building jar: /Users/ joanbarcelo /NetBeansProjects/P1/dist/P1.jar

To run this application from the command line without Ant, try:

java -jar "/Users/joanbarcelo/NetBeansProjects/P1/dist/P1.jar"

deploy:

jar:

BUILD SUCCESSFUL (total time: 0 seconds)

Una vez hecha la prueba de la clase sumador, le quitamos el main, y crearemos una clase que sea capaz de lanzar varios procesos. La clase Sumador se quedará así:

```
public class Sumador {
        public int sumar(int n1, int n2){
                int resultado=0;
                for (int i=n1;i<=n2;i++){
                        resultado=resultado+i;
                }
                return resultado;
        }
}
```

Y ahora tendremos una clase que lanza procesos de esta forma:

```java
package com.ies;

public class Lanzador {
	public void lanzarSumador(Integer n1,
			Integer n2){
		String clase="com.ies.Sumador";
		ProcessBuilder pb;
		try {
			pb = new ProcessBuilder(
					"java",clase,
					n1.toString(),
					n2.toString());
			pb.start();
		} catch (Exception e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
	}
	public static void main(String[] args){
		Lanzador l=new Lanzador();
		l.lanzarSumador(1, 51);
		l.lanzarSumador(51, 100);
		System.out.println("Ok");
	}
}
```

Este ejemplo superior se ha realizado con la clase ProcessBuilder, java tiene otra clase para la creación de procesos llamada Runtime, el proceso y código para la ejecución de procesos con Runtime se realiza de la siguiente forma:

Runtime rt = Runtime.getRuntime();

String ruta = "java -jar /Users/jaumecampsfornari/NetBeansProjects/P1/dist/P1.jar 3 7";

Process p = rt.exec(ruta);

5. Comunicación entre procesos.

Las operaciones multiproceso pueden implicar que sea necesario comunicar información entre muchos procesos, lo que obliga a la necesidad de utilizar mecanismos específicos de comunicación que ofrecerá Java o a diseñar alguno separado que evite los problemas que puedan aparecer.

Un proceso es un programa en ejecución y, como cualquier programa, recibe información, la transforma y produce resultados. Esta acción se gestiona a través de:

- La entrada estándar (stdin): Lugar de donde el proceso lee los datos de entrada que requiere para su ejecución. Normalmente suele ser el teclado, pero podría recibirlos de un fichero, de la tarjeta de red o fines de otro proceso, entre otros lugares. La lectura de datos a lo largo de un programa leerá los datos de su entrada estándar.

- La salida estándar (stdout): Lugar donde el proceso escribe los resultados que obtiene. Normalmente es la pantalla, a pesar de que podría ser, entre otras, la impresora o hasta otro proceso que necesite estos datos como entrada. La escritura de datos que se realice en un programa (por ejemplo intermediando printf en C o System.out.println en Java) se produce por la salida estándar.

- La salida de error (stderr): Lugar donde el proceso envía los mensajes de error. Habitualmente es el mismo que la salida estándar, pero puede especificar que sea otro lugar, por ejemplo un fichero para identificar más fácilmente los errores que tienen lugar durante la ejecución. La utilización de System.out y System.err en Java se puede ver como un ejemplo de utilización de estas salidas.

En el ejemplo, el segundo proceso suele sobrescribir el resultado del primero, así que modificaremos el código del lanzador para que cada proceso use su propio fichero de resultados.

```
public class Lanzador {
	public void lanzarSumador(Integer n1,
			Integer n2, String fichResultado){
		String clase="com.ies.Sumador";
		ProcessBuilder pb;
		try {
			pb = new ProcessBuilder(
					"java",clase,
					n1.toString(),
					n2.toString());

			pb.redirectError(new File("errores.txt"));
			pb.redirectOutput(new File(fichResultado));
			pb.start();
		} catch (Exception e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}
	}
	public static void main(String[] args){
		Lanzador l=new Lanzador();
		l.lanzarSumador(1, 5, "result1.txt");
		l.lanzarSumador(6,10, "result2.txt");
		System.out.println("Ok");
	}
}
```

Usando el método .directory(new File("c:\\dir\\)) se puede indicar a Java donde está el archivo que se desea ejecutar.

6. Gestión de procesos.

- El sistema operativo es el encargado de crear los nuevos procesos siguiendo las directrices del usuario.
- Cuando un usuario quiere abrir un programa, el sistema operativo es el responsable de crear y posar en ejecución el proceso correspondiente que lo ejecutará.
- La puesta en ejecución de un nuevo proceso se produce a causa de que hay un proceso en concreto que está pidiendo su creación.
- Cualquier proceso en ejecución siempre depende del proceso que lo creó, estableciendo un vínculo entre los dos. A su vez, el nuevo proceso puede crear nuevos procesos, formándose el que se denomina un árbol de procesos.

La gestión de procesos se realiza de dos formas muy distintas en función de los dos grandes sistemas operativos: Windows y Linux.

En Windows toda la gestión de procesos se realiza desde el «Administrador de tareas» al cual se accede con Ctrl+Alt+Supr o bien desde el CMD usando el comando “tasklist”. En GNU/Linux se puede utilizar un terminal de consola para la gestión de procesos. Desde ambas, no solo se pueden arrancar procesos si no también detenerlos, reanudarlos, terminarlos y modificar su prioridad de ejecución, con el comando “top” podemos ver los procesos actuales del sistema.

Para identificar los procesos, los sistemas operativos suelen utilizar un identificador de proceso (process identifier [PID]) unívoco para cada proceso. La utilización del PID es básica en la hora de gestionar procesos, puesto que es la forma que tiene el sistema de referirse a los procesos que gestiona.

7. Comandos para la gestión de procesos en sistemas libres y propietarios.

En sistemas Windows, los procesos se pueden ejecutar, reiniciar o parar desde interfaz gràfica, a través del administrador de tareas o la ruta de “Servicios” en servidores Windows. Además de la interface gráfica, también se pueden gestionar a través de la consola CMD.

- Tasklist (Muestra los procesos).
- Taskkill /PID 26356 /F (Matar un procesos “F para forzar el parado”).
-

https://tweaks.com/windows/39559/kill-processes-from-command-prompt/

En GNU/Linux se puede utilizar un terminal de consola para la gestión de procesos, lo que implica que no solo se pueden arrancar procesos si no también detenerlos, reanudarlos, terminarlos y modificar su prioridad de ejecución.

- Para arrancar un proceso, simplemente tenemos que escribir el nombre del comando correspondiente. Desde GNU/Linux se pueden controlar los servicios que se ejecutan con un comando llamado service. Por ejemplo, se puede usar sudo service apache2 stop para parar el servidor web y sudo service apache2 start para volver a ponerlo en marcha. También se puede reiniciar un servicio (tal vez para que relea un fichero de configuración que hemos cambiado) con sudo service apache2 restart.
- Se puede detener y/o terminar un proceso con el comando kill. Se puede usar este comando para terminar un proceso sin guardar nada usando kill -SIGKILL <numproceso> o kill -9 <numproceso>. Se puede pausar un proceso con kill -SIGSTOP <numproceso> y rearrancarlo con kill -SIGCONT
- Se puede enviar un proceso a segundo plano con comandos como bg o al arrancar el proceso escribir el nombre del comando terminado en &.
- Se puede devolver un proceso a primer plano con el comando fg.

7.1. Prioridades

En sistemas como GNU/Linux se puede modificar la prioridad con que se ejecuta un proceso. Esto implica dos posibilidades

- Si pensamos que un programa que necesitamos ejecutar es muy importante podemos darle más prioridad para que reciba «más turnos» del planificador.
- Y por el contrario, si pensamos que un programa no es muy necesario podemos quitarle prioridad y reservar «más turnos de planificador» para otros posibles procesos.

El comando nice permite indicar prioridades entre -20 y 19. El -20 implica que un proceso reciba la máxima prioridad, y el 19 supone asignar la mínima prioridad.

7.2. Sincronización entre procesos

Diferentes códigos o procesos pueden consistir en los mismos recursos variables u otros que necesitan ser leídos o escritos, pero cuyos resultados dependerá del orden en que se producen las acciones. Por ejemplo, si una variable 'x' es para ser leído por el procedimiento A y procedimiento B tiene que escribir en la misma variable 'x' al mismo tiempo, el proceso A puede obtener ya sea el valor antiguo o nuevo de 'x'.

Proceso A:

b = x + 5

Proceso B:

x = 3 + z

Si dos o más procesos avanzan por esta sección de código es perfectamente posible que unas veces nuestro programa multiproceso se ejecute bien y otras no.

En todo programa multiproceso pueden encontrarse estas zonas de código «peligrosas» que deben protegerse especialmente utilizando ciertos mecanismos. El nombre global para todos los lenguajes es denominar a estos trozos «secciones críticas».

7.3. Mecanismos para controlar secciones críticas

Los mecanismos más típicos son los ofrecidos por UNIX/Windows:

- Semáforos.
- Colas de mensajes.
- Tuberías (pipes)
- Bloques de memoria compartida.

En realidad algunos de estos mecanismos se utilizan más para intercomunicar procesos, aunque para los programadores Java la forma de resolver el problema de la «sección crítica» es más simple.

8. Sincronización entre procesos.

Los métodos de comunicación de procesos se pueden considerar como métodos de sincronización puesto que permiten al proceso padre llevar el ritmo de envío y recepción de mensajes.
Concretamente, los envíos y recepciones por los flujos de datos permiten a un proceso hijo comunicarse con su padre a través de un canal de comunicación unidireccional bloqueando.

Espera de procesos (operación wait):

- Además de la utilización de los flujos de datos se puede esperar por la finalización del proceso hijo y obtener su información de finalización mediante la operación wait.
- La mencionada operación bloquea al proceso padre hasta que el hijo finaliza su ejecución mediante exit. Como resultado se recibe la información de finalización del proceso hijo.

Espera de procesos (operación wait):

- Este valor de retorno se especifica mediante un número entero. El valor de retorno indica el resultado de la ejecución. No tiene nada que ver con los mensajes que se pasan entre padre e hijo a través de los streams. Por convención se utiliza 0 para indicar que el hijo ha acabado de forma correcta.

- Con waitFor() de la clase Process el padre espera bloqueado hasta que el hijo finaliza su ejecución, volviendo inmediatamente si el hijo ha finalizado con anterioridad o si alguien lo interrumpe (en este caso se lanza la interrupción InterruptedException).
- Además se puede utilizar exitValue() para obtener el valor de retorno que volvió un proceso hijo. El proceso hijo tiene que haber finalizado, si no, se lanza la excepción IllegalThreadStateException.

9. Depuración.

¿Como se depura un programa multiproceso/multihilo? Por desgracia puede ser muy difícil:

a) No todos los depuradores son capaces.
b) A veces cuando un depurador interviene en un proceso puede ocurrir que el resto de procesos consigan ejecutarse en el orden correcto y dar lugar a que el programa parezca que funciona bien.
c) Un error muy típico es la NullPointerException, que en muchos casos se deben a la utilización de referencias Java no inicializadas o incluso a la devolución de valores NULL que luego no se comprueban en alguna parte del código.
d) Se puede usar el método redirectError pasándole un objeto de tipo File para que los mensajes de error vayan a un fichero.
e) Se debe recordar que la «visión» que tiene Eclipse del sistema puede ser muy diferente de la visión que tiene el proceso lanzado. Un problema muy común es que el proceso lanzado no encuentre clases, lo que obligará a indicar el CLASSPATH.
f) Un buen método para determinar errores consiste en utilizar el entorno de consola para lanzar comandos para ver «como es el sistema» que ve un proceso fuera del IDE.

En general todos los fallos en un programa multiproceso vienen derivado de no usar synchronized de la forma correcta.