Programación multiproceso II

Programación de servicios y procesos

Contenidos:

1) Gestión de procesos. Conceptos básicos: Creación, ejecución y finalización de procesos.
2) **Sincronización entre procesos. Exclusión mutua. Condiciones de sincronización.**
3) Recursos compartidos.
4) Mecanismos de comunicación y sincronización de procesos (semáforos, monitores, paso de mensajes.)
5) Problemas. Inanición, interbloqueos.

Sincronización de procesos. Introducción.

En un sistema multiprogramado con un único procesador, los procesos se intercalan en el tiempo aparentando una ejecución simultánea. Aunque no se logra un procesamiento paralelo y produce una sobrecarga en los intercambios de procesos, la ejecución intercalada produce beneficios en la eficiencia del procesamiento y en la estructuración de los programas.

La intercalación y la superposición pueden contemplarse como ejemplos de procesamiento concurrente en un sistema monoprocesador, los problemas son consecuencia de la velocidad de ejecución de los procesos que no pueden predecirse y depende de las actividades de otros procesos, de la forma en que el sistema operativo trata las interrupciones surgen las siguientes dificultades:

- Compartir recursos globales es arriesgado.
- Para el sistema operativo es difícil gestionar la asignación óptima de recursos.

Las dificultades anteriores también se presentan en los sistemas multiprocesador.

El hecho de compartir recursos ocasiona problemas, por esto es necesario proteger a dichos recursos. Los problemas de concurrencia se producen incluso cuando hay un único procesado

Sincronización de procesos. Labores del S.O.

Elementos de gestión y diseño que surgen por causa de la concurrencia:

- El sistema operativo debe seguir a los distintos procesos activos
- El sistema operativo debe asignar y retirar los distintos recursos a cada proceso activo, entre estos se incluyen:
  - Tiempo de procesador
  - Memoria
  - Archivos
  - Dispositivos de E/S
- El sistema operativo debe proteger los datos y los recursos físicos de cada proceso contra injerencias no intencionadas de otros procesos.
- Los resultados de un proceso deben ser independientes de la velocidad a la que se realiza la ejecución de otros procesos concurrentes.

Sincronización de procesos. Labores del S.O.

El SO establece un orden de ejecución para los procesos, esta planificación es llevada a cabo mediante un algoritmo de planificación (**scheduler**). Podemos clasificar los algoritmo de planificación en dos tipos:

- **Apropiativos**: Bajo ciertas condiciones, los procesos pueden ser expulsados de la CPU. El uso de este tipo de algoritmos implica:
  - Mayor coste de implementación.
  - Mejor optimización del uso de la CPU.
  - Utilizados en sistemas multiusuario y multitarea.

- **No apropiativos**: Una vez el proceso es ejecutado por la CPU, esta adquiere su control hasta que terminan o se bloquean. Sus características principales:
  - Sencillos.
  - Rendimiento menor.
  - Utilizados en sistemas batch o monousuario.

Sincronización de procesos.

Ateniendo a la capacidad de comunicación y sincronización con otros procesos y recursos del sistema, podemos distinguir los siguientes tipos de procesos:

- **Independientes**: Son los menos frecuentes, no se comunican con ningún otro proceso.
- **Cooperativos**: Se comunican y cooperan con el resto de procesos para realizar una tarea común.
- **Competitivos**: Hacen uso de lo/s mismo/s recursos/s, por lo tanto, lucharán por este/os.

En el segundo y tercer caso, necesitamos componentes que nos permitan establecer acciones de **sincronización** y **comunicación** entre los procesos.

Un proceso entra en **condición de competencia** con otro, cuando **ambos necesitan el mismo recurso, ya sea forma exclusiva o no;** por lo que será necesario utilizar mecanismos de sincronización y comunicación entre ellos.

Sincronización de procesos. Exclusión mútua.

En el caso de procesos competidores, vamos a comenzar viendo unas definiciones:

- Cuando un proceso necesita un recurso de forma **exclusiva**, es porque mientras que lo esté utilizando él, ningún otro puede utilizarlo. Se llama **región de exclusión mutua o región crítica** al **conjunto de instrucciones en las que el proceso utiliza un recurso y que se deben ejecutar de forma exclusiva con respecto a otros procesos competidores por ese mismo recurso.**

- Cuando más de un proceso necesitan el mismo recurso, antes de utilizarlo tienen que pedir su uso, una vez que lo obtienen, el resto de procesos quedarán bloqueados al pedir ese mismo recurso. Se dice que un proceso hace **un lock (bloqueo) sobre un recurso cuando ha obtenido su uso en exclusión mutua**.

- Por ejemplo dos procesos, compiten por dos recursos distintos, y ambos necesitan ambos recursos para continuar. Se puede dar la situación en la que cada uno de los procesos bloquee uno de los recursos, lo que hará que el otro proceso no pueda obtener el recurso que le falta; quedando bloqueados un proceso por el otro sin poder finalizar. **Deadlock o interbloqueo, se produce cuando los procesos no pueden obtener, nunca, los recursos necesarios para continuar su tarea**. El interbloqueo es una situación muy peligrosa, ya que puede llevar al sistema a su caída o cuelgue.

Comunicación entre procesos.

Comunicación entre procesos: un proceso da o deja información; recibe o recoge información.

Los lenguajes de programación y los sistemas operativos, nos proporcionan **primitivas** de sincronización que facilitan la interacción entre procesos de forma sencilla y eficiente.

Una **primitiva**, hace referencia a una operación de la cual conocemos sus restricciones y efectos, pero no su implementación exacta. Veremos que usar esas **primitivas** se traduce en utilizar **objetos** y sus **métodos**, teniendo muy en cuenta sus repercusiones reales en el comportamiento de nuestros procesos.

Clasificaremos las interacciones entre los procesos y el resto del sistema (recursos y otros procesos), como estas tres:

- **Sincronización**: Un proceso puede conocer el punto de ejecución en el que se encuentra otro en ese determinado instante.
- **Exclusión mutua**: Mientras que un proceso accede a un recurso, ningún otro proceso accede al mismo recurso o variable compartida.
- **Sincronización condicional**: Sólo se accede a un recurso cuando se encuentra en un determinado estado interno.

Mecanismos de comunicación y sincronización de procesos

Si pensamos en la forma en la que un proceso puede comunicarse con otro. Se nos ocurrirán estas dos:

- **Intercambio de mensajes**: Tendremos las primitivas **enviar** (send) y **recibir** (receive o wait) información.
- **Recursos (o memoria) compartidos:** Las primitivas serán **escribir** (write) y **leer** (read) datos en o de un recurso.

El intercambio de mensajes, se puede realizar de dos formas:

- ➔ Utilizar un **buffer de memoria**.
- ➔ Utilizar un **socket**.

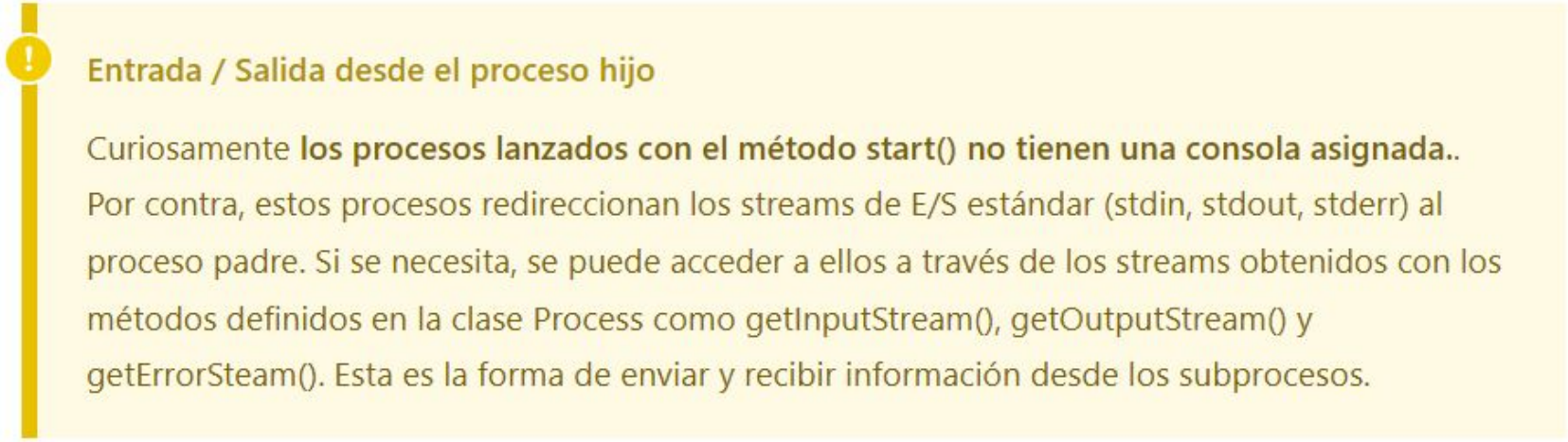
La diferencia entre ambos, está en que un socket se utiliza para intercambiar información entre procesos en distintas máquinas a través de la red; y un buffer de memoria, crea un canal de comunicación entre dos procesos utilizando la memoria principal del sistema. Actualmente, es más común el uso de sockets que buffers para comunicar procesos.

En java, utilizaremos **sockets** y **buffers** como si utilizáramos cualquier otro **stream o flujo de datos**. Utilizaremos los métodos read-write en lugar de send-receive.

Comunicación de procesos con Java

La clase **Process** es una clase abstracta definida en el paquete java.lang y contiene la información del proceso en ejecución. Tras invocar al método **start** de **ProcessBuilder**, éste devuelve una referencia al proceso en forma de objeto Process.

Los métodos de la clase Process pueden ser usados para realizar operaciones de E/S desde el proceso, para comprobar su estado, su valor de retorno, para esperar a que termine de ejecutarse y para forzar la terminación del proceso. Sin embargo estos métodos no funcionan sobre procesos especiales del SO como pueden ser servicios, shell scripts, demonios, etc.



Comunicación de procesos con Java

Ya hemos hablado anteriormente de las clases Process y ProcessBuilder. La clase **Process** proporciona métodos para realizar la entrada desde el proceso, para obtener la salida, esperar a que el proceso se complete, comprobar el estado de la salida y destruirlo.

Tabla 1.2. Métodos de la clase java.lang.Process

Método	Descripción
destroy()	Destruye el proceso sobre el que se ejecuta.
exitValue()	Devuelve el valor de retorno del proceso cuando este finaliza. Sirve para controlar el estado de la ejecución.
getErrorStream()	Proporciona un InputStream conectado a la salida de error del proceso.
getInputStream()	Proporciona un InputStream conectado a la salida normal del proceso.
getOutputStream()	Proporciona un OutputStream conectado a la entrada normal del proceso.
isAlive()	Determina si el proceso está o no en ejecución.
waitFor()	Detiene la ejecución del programa que lanza el proceso a la espera de que este último termine.

Comunicación de procesos con Java

- Los métodos de comunicación de procesos se pueden considerar como métodos de sincronización ya que permiten al proceso padre llevar el ritmo de envío y recepción de mensajes.

- Además de la utilización de los flujos de datos se puede esperar por la finalización del proceso hijo mediante la operación **waitFor()**.
  - Bloquea al proceso padre hasta que el hijo finaliza su ejecución mediante exit.
  - Como resultado el padre recibe la información de finalización del proceso hijo.
    - El valor de retorno se especifica mediante un número entero, cómo resultado de la ejecución.
    - No tiene nada que ver con los mensajes que se pasan padre e hijo a través de los streams.
    - Por convención se utiliza 0 para indicar que el hijo ha acabado de forma correcta.

Comunicación de procesos con Java

Boolean waitFor(long timeOut, TimeUnit unit)	El funcionamiento es el mismo que en el caos anterior sólo que en esta ocasión podemos especificar cuánto tiempo queremos esperar a que el proceso hijo termine. El método devuelve true si el proceso termina antes de que pase el tiempo indicado y false si ha pasado el tiempo y el proceso no ha terminado.

```
// Wait for the process to end for 10 seconds.
if (p.waitFor(10, TimeUnit.SECONDS)) {
    System.out.println("Process has finished");
} else {
    System.out.println("Timeout. Process hasn't finished");
}
// Force process termination.
p.destroy();
```

Códigos de terminación

Un código de salida (exit code o a veces también return code) es el valor que un proceso le devuelve a su proceso padre para indicarle cómo ha acabado. Si un proceso acaba con un valor de finalización 0 es que todo ha ido bien, cualquier otro valor entre 1 to 255 indica alguna causa de error.

Comunicación de procesos con Java

Por defecto el proceso que se crea no tiene su propia terminal o consola. Todas las operaciones E/S serán redirigidas al proceso padre, donde se puede acceder a ellas con los métodos **getInputStream(), getOutpuStream() y getErrorStream()**. El proceso utiliza estos flujos para alimentar la entrada y obtener la salida del proceso.

Cada constructor de **ProcessBuilder** gestiona los siguientes atributos del proceso:

- Un **comando**. En una lista de cadenas que representa el programa y los argumentos (si los hay).
- Un **entorno** (*environment*) con sus variables
- Un **directorio** de trabajo (por defecto el del proceso en curso)
- Una **fuente** o tubería estándar de **entrada**. Se accede a ella por el método **Process.getOutputStream()**. Esta fuente puede ser redirigida con el método **redirectInput()** (si lo hacemos "vaciamos" el stream y el método Process.getOutputStream() devolverá nulo).
- Una **fuente** o tubería estándar de **salida** y otra de **error**. Se accede a ellas por los métodos **Process.getInputStream()** y **Process.getErrorStream()**. Estas fuentes pueden ser redirigidas con los métodos **redirectOutput()** y **redirectError()** (si lo hacemos "vaciamos" el stream correspondiente).

Comunicación de procesos con Java

Para las actividades os pediré que programéis tanto el proceso padre como el/los proceso/s hijo/s. Para hacer eso, una de las clases tendrá que lanzar a la/s otras.

Cada proceso se programa como un proyecto independiente y en cada proyecto debe haber, una clase con el método main.

Antes de que una clase pueda lanzar a otra, al menos la segunda (proceso hijo) debe estar compilada. El proyecto padre además debe tener la ruta de invocación al proyecto hijo.

Comunicación de procesos con Java - getInputStream/getErrorStream/ getOutputStream

No sólo es importante recoger el valor de retorno de un comando, sino que muchas veces nos va a ser de mucha utilidad el poder obtener la información que el proceso genera por la salida estándar o por la salida de error.

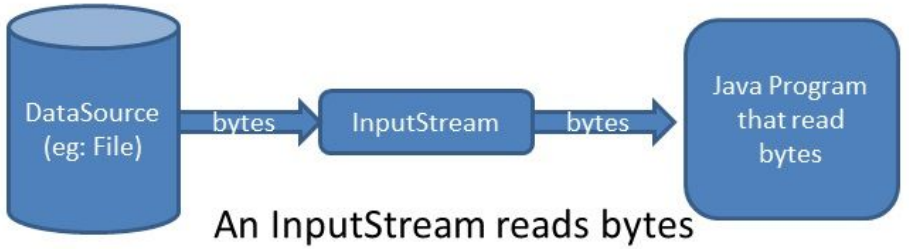
Para esto vamos a utilizar el método **getInputStream()** de la clase Process para leer el stream de salida del proceso, es decir, para leer lo que el comando ejecutado (proceso hijo) ha enviado a la consola.

Curiosamente, o no tanto, además de la salida estándar, también podemos obtener la salida de error que genera el proceso hijo para procesarla desde el padre. Lo haremos utilizando el método **getErrorStream()**.

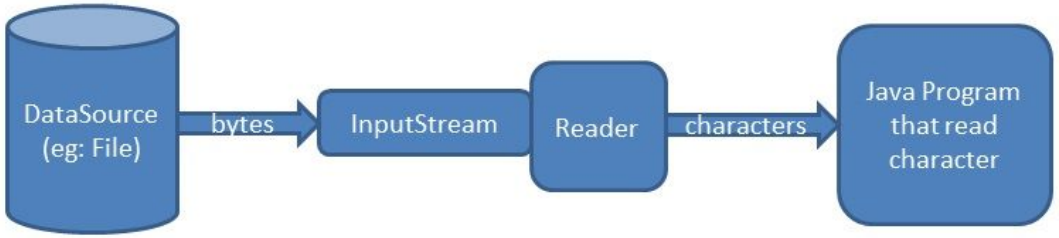
No sólo podemos recoger la información que envía el proceso hijo sino que, además, también podemos enviar información desde el proceso padre al proceso hijo, usando el último de los tres métodos **getOutputStream()**.

Aunque hay diferentes clases y formas de tratar los los streams de entrada, yo en general voy a optar por utilizar la clase **Scanner** que es la que me “envuelve” el stream y me permite manejar la información de forma más sencilla.

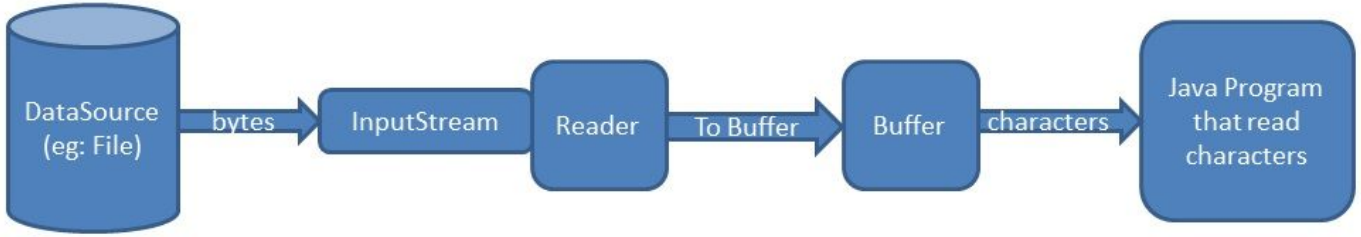
Comunicación de procesos con Java - getInputStream()



An InputStream reads bytes



A Reader works with an InputStream and reads characters



A BufferedReader pre-fetches characters in to a buffer for performance

Comunicación de procesos con Java. Ejemplos

- Ejemplo de clase Java recibe una entrada y valida si lo que llega es un número y si es par o impar => 9_Calcular.java (proyecto CalcularParImpar)

- Ejemplo de clase Java que llama a un Jar del proyecto anterior (Ejemplo1) => 7_LlamarEjemploDirectorio.java

Comunicación de procesos con Java. Ejemplos

2.- Programa de Calcular

2.- Leemos el Stream de entrada

```
Scanner sc = new Scanner(System.in);
Integer numero;

try {

    //numero = Integer.parseInt(br.readLine());
    numero = sc.nextInt();
    if (numero % 2 == 0) {
        System.out.println("El número es par" );
    } else {
        System.out.println("El número es impar" );
        System.out.println("El número es impar REPITO" );
    }
    System.exit(0);
    }
catch (Exception e) {
    e.printStackTrace();
    System.exit(-1);
    }
```

1.- Asignamos a un Stream de datos la entrada del proceso.

3.- Si podemos hacer la validación, salimos con OK, Si hay error, salimos con KO

Comunicación de procesos con Java. Ejemplos

Dentro de 9_Pedir.java

```
//creamos objeto File al directorio donde esta Ejemplo2
File directorio = new File("C:\\Users\\jhorn\\eclipse-workspace\\UT2\\UT2_9_CalcularParImpar\\bin");

//El proceso a ejecutar es Ejemplo2
ProcessBuilder pb = new ProcessBuilder("java", "mispaquetes.Calcular");

//se establece el directorio donde se encuentra el ejecutable
pb.directory(directorio);

//se ejecuta el proceso
Process p = pb.start();
```

0.- Indicamos la ruta del programa que queremos ejecutar.

Indicamos que queremos ejecutar un programa java y el paquete y clase (ue al ser un proyecto Java debe tener un main)

Comunicación de procesos con Java. Ejemplos

Dentro de 9_Pedir.java

```
System.out.println("Escribe un número:" );
Scanner sc = new Scanner(System.in);
String respuesta = sc.nextLine();
…..
```

1.- Pedimos por consola un número y lo capturamos

```
respuesta = respuesta+"\n";
OutputStream os = p.getOutputStream();
os.write(respuesta.getBytes());
os.flush(); // vacía el buffer de salida
```

2.- Asignamos la respuesta al Stream de entrada. Es imprescindible ponerle el salto de linea

Comunicación de procesos con Java. Ejemplos

Dentro de 9_Pedir.java

```
// COMPROBACION DE ERROR - 0 bien - 1 mal
int exitVal = -1;
try {
    exitVal = p.waitFor();
    System.out.println("Valor de Salida: " + exitVal);
} catch (InterruptedException e) {
    e.printStackTrace();
}

if (exitVal == 0) {
   //obtener la salida devuelta por el proceso
    try {
        InputStream is = p.getInputStream();
        Scanner sc2 = new Scanner(is);
        if(sc2.hasNext() == true) {
            String salida = sc2.nextLine();
            System.out.println(salida);
        }
        is.close();
    } catch (Exception e) {
        e.printStackTrace();
    }
}
```

1.- Siempre conviene comprobar si todo a ido bien para actuar en consecuencia y normalmente condicionar los siguientes pasos a este valor

2.- Recogemos la salida

3.- La pintamos, como también podríamos guardarla en un fichero por ejemplo

Comunicación de procesos con Java. Ejemplos

- Ejemplo de clase Java que llama a un proceso y lee el Stream de salida => 6_EjemploDirectorio.java (proyecto Ejemplo1)

- Ejemplo de clase Java que llama a un Jar del proyecto anterior (Ejemplo1) =>
- 7_LlamarEjemploDirectorio.java

Comunicación de procesos con Java. Ejemplos

Dentro de 7_LlamarEjemploDirectorio.java y una vez lanzado el proceso

```
 // COMPROBACIÓN DE ERROR - 0 bien - 1 mal
...

 //obtener la salida devuelta por el proceso
      try {
          InputStream is = p.getInputStream();
          int c;
          while ((c = is.read()) != -1)
                System.out.print((char) c);
          is.close();
      } catch (Exception e) {
          e.printStackTrace();
      }
```

1.- Siempre conviene comprobar si todo a ido bien para actuar en consecuencia y normalmente condicionar los siguientes pasos a este valor

2.- Recogemos la salida

3.- La pintamos, como también podríamos guardarla en un fichero por ejemplo

Comunicación de procesos con Java. Ejemplos

Dentro de 6_EjemploDirectorio.java

```
public static void main(String[] args) throws IOException {
        ProcessBuilder pb = new ProcessBuilder("CMD", "/C", "DIR");
        pb.directory(new File("c:/Julio/"));
        Process p = pb.start();
        try {
              InputStream is = p.getInputStream();
              // mostramos en pantalla caracter a caracter
               int c;
               while ((c = is.read()) != -1)
                     System.out.print((char) c);
               is.close();
```

1.- Vamos a lanzar la consola de Windows y hacer un DIR

2.- Indicamos el directorio

3.- Capturamos en un stream el flujo de datos de salida

4.- “grabamos” en el flujo de datos de salida, la salida que queremos

Comunicación de procesos con Java. Ejercicio

Crea un programa Java llamado *LeerNombre.java* que reciba desde los argumentos del *main()* un nombre y lo visualice en pantalla. Utiliza el valor 1 para la salida cuando todo ha ido correctamente y el valor -1 para la salida cuando algo falla.

A continuación crea otro programa llamado *LlamarLeerNombre.java* para ejecutar *LeerNombre.java*.

Utiliza el método *waitFor()* para comprobar el valor de la salida del proceso que se ejecuta. Prueba el programa ejecutando la llamada pasando un nombre como argumento y sin pasarlo.