Programación de Procesos y Servicios

UT1: Programación de Procesos
2º DAM

Relación padre-hijo

Cuando un proceso crea otro, se establece una relación de padre-hijo. Ambos se ejecutan concurrentemente y comparten la CPU según la política del sistema operativo.

- Creación: Un proceso crea otro, estableciendo una relación padre-hijo.
- Ejecución: Ambos se ejecutan concurrentemente y comparten la CPU según la política del SO.
- Memoria: Aunque tienen un vínculo, padre e hijo son independientes y tienen espacios de memoria separados.
- Sincronización: El padre puede esperar al hijo con *wait* y terminarlo con *destroy*. El proceso finaliza con *exit*.

Procesos en Java

Cuando trabajamos con procesos en Java, hay que tener en cuenta las siguientes características:

- Ejecución Asíncrona: Padre e hijo no tienen que ejecutarse concurrentemente.
- Sin Terminación en Cascada: Los hijos pueden sobrevivir al padre y continuar ejecutándose asíncronamente.

En el paquete *java.lang* tenemos dos clases para la gestión de procesos.

- *java.lang.ProcessBuilder*
- *java.lang.Process*

ProcessBuilder

Las instancias de **ProcessBuilder** configuran los atributos de un proceso antes de su ejecución. Es una clase auxiliar que permite ajustar parámetros clave para la ejecución del proceso.

```
ProcessBuilder(String... command)
```

Crea una nueva instancia de ProcessBuilder con el comando y sus argumentos especificados. Cada argumento del comando se pasa como un string en el array *command*. Este comando será el que se ejecutará cuando se inicie el proceso.

```
ProcessBuilder pb = new ProcessBuilder("ping", "google.com");
```

ProcessBuilder

directory(File file)

Establece el directorio de trabajo en el que se ejecutará el proceso. Por defecto, el directorio de trabajo actual se establece en el valor devuelto por la propiedad del sistema *user.dir*.

```
ProcessBuilder pb = new ProcessBuilder("ping", "google.com");
pb.directory(new                                File("/user/home/"));
```

directory()

Devuelve el directorio de trabajo que se ha configurado para el proceso.

```
ProcessBuilder pb = new ProcessBuilder("myCommand");
File dir = pb.directory();
```

ProcessBuilder

environment()

Devuelve un mapa de las variables de entorno que se usarán al iniciar el proceso. Este mapa permite acceder, modificar o agregar variables de entorno que estarán disponibles para el proceso.

```
ProcessBuilder pb = new ProcessBuilder("ls", "MY_DIR");
Map<String, String> env = pb.environment();
env.put("MY_DIR", "/usr/home/luis_butron");
```

ProcessBuilder

`inheritIO()`

Permite que el proceso hijo herede los flujos de entrada, salida y error del proceso padre. Esto significa que la entrada estándar (System.in), la salida estándar (System.out), y el flujo de error estándar (System.err) del proceso hijo estarán vinculados directamente a los del proceso padre, compartiendo la misma consola o terminal.

```
ProcessBuilder pb = new ProcessBuilder("myCommand");
pb.inheritIO();
```

ProcessBuilder

`start()`

Inicia la ejecución del proceso con la configuración previamente establecida. Al llamar a *start()*, se lanza el proceso en un nuevo subproceso (hijo), devolviendo una instancia de la clase Process, que permite controlar y monitorear la ejecución del proceso.

```
ProcessBuilder pb = new ProcessBuilder("touch", "hola.txt");
Process process = pb.start();
```

Process

La clase **Process** en Java representa un proceso en ejecución. Proporciona métodos para interactuar con el proceso, gestionar su entrada y salida, y esperar su finalización.

destroy()

Finaliza un proceso.

```
Process process = new ProcessBuilder("myCommand").start();
process.destroy();
```

Process

`waitFor()`

Bloquea el hilo actual hasta que el proceso asociado haya terminado su ejecución. Este método se utiliza para esperar a que el proceso termine y luego obtener su código de salida.

- Bloqueo: El método bloquea el hilo que lo invoca hasta que el proceso finalice. Esto asegura que el programa no continúe hasta que el proceso haya terminado.
- Excepciones: Puede lanzar *InterruptedException* si el hilo que está esperando se interrumpe mientras está bloqueado.

```
Process process = new ProcessBuilder("myCommand").start();
int exitCode = process.waitFor();
System.out.println("El código de salida es: " + exitCode);
```

Process

waitFor(long timeout, TimeUnit unit)

Espera a que el proceso termine, pero con un límite de tiempo especificado. Esto significa que el método bloqueará el hilo actual hasta que el proceso finalice o hasta que se alcance el tiempo de espera, lo que ocurra primero.

- Retorno: Devuelve un valor *booleano*. Retorna *true* si el proceso terminó dentro del tiempo de espera especificado, y *false* si el tiempo de espera se agotó antes de que el proceso terminara.
- Excepciones: Puede lanzar *InterruptedException* si el hilo que está esperando se interrumpe mientras está bloqueado.

```
Process process = new ProcessBuilder("myCommand").start();
boolean finished = process.waitFor(60, TimeUnit.SECONDS);
if (finished) {
      System.out.println("El proceso terminó dentro del tiempo de espera.");
} else {
      System.out.println("El tiempo de espera se agotó antes de que el proceso terminara.");
}
```

Process

`exitValue()`

Devuelve el código de salida del proceso una vez que ha terminado. El código de salida es un valor entero que el proceso devuelve al sistema operativo al finalizar, indicando el resultado de su ejecución.

- Excepciones: Lanza una excepción *IllegalThreadStateException* si el proceso aún está en ejecución cuando se llama a este método. Esto significa que debes asegurarte de que el proceso haya terminado antes de llamar a *exitValue()*.

```
Process process = new ProcessBuilder("myCommand").start();
int exitCode = process.waitFor();
int code = process.exitValue();
```

Process

isAlive()

Verifica si el proceso asociado aún está en ejecución.

```
Process process = new ProcessBuilder("myCommand").start();
boolean isRunning = process.isAlive();
if (isRunning) {
    System.out.println("El proceso sigue en ejecución.");
} else {
    System.out.println("El proceso ha terminado.");
}
```

Process

Resumen

Método	Descripción
`int exitValue()`	Código de finalización devuelto por el proceso hijo
`Boolean isAlive()`	Comprueba si el proceso todavía está en ejecución
`int waitFor()`	Hace que el proceso padre se quede esperando a que el proceso hijo termine. Devuelve el código de finalización del proceso hijo
`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.
`void destroy()`	Estos dos métodos se utilizan para matar al proceso. El segundo lo hace de forma forzosa.

Gestión de la E/S

Un subproceso no tiene terminal o consola en el que poder mostrar su información. Toda la E/S por defecto por defecto se redirige al proceso padre. Es el proceso padre el que puede usar estos streams para recoger o enviar información al proceso hijo.

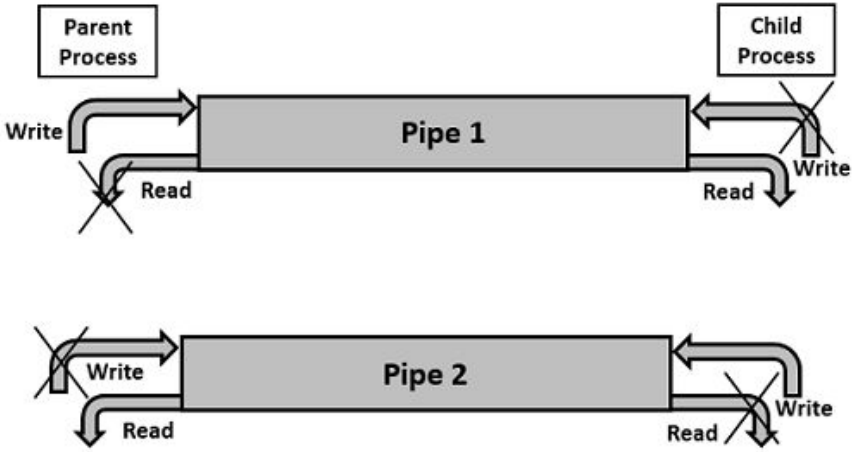
En sistemas Linux, la entrada/salida (E/S) se maneja como si fueran archivos. Cada proceso tiene tres descriptores de archivo predeterminados:

- 0 para la entrada estándar (stdin)
- 1 para la salida estándar (stdout)
- 2 para los errores estándar (stderr).

Estos descriptores permiten redirigir y gestionar la entrada y salida de datos de diversas maneras, como redirigir la salida a un archivo o leer la entrada desde un archivo en lugar del teclado.

Gestión de la E/S

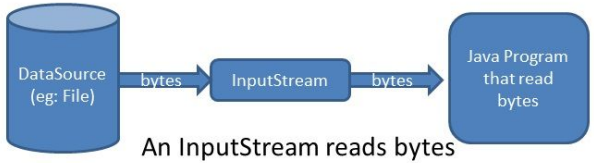
En la relación padre-hijo que se crea entre procesos los descriptores también se redirigen desde el hijo hacia el padre, usando 3 tuberías (pipes), una por cada stream de E/S por defecto. Esas tuberías pueden usarse de forma similar a cómo se hace en los sistemas Linux.



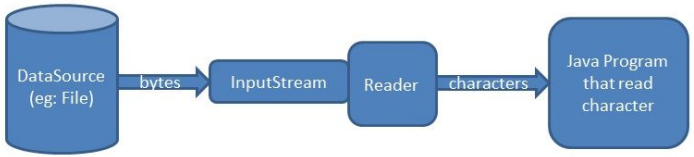
Gestión de la E/S

getInputStream()

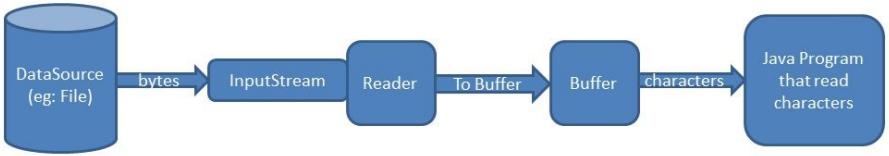
Este método de la clase *Process* nos permite obtener el stream de salida del proceso, es decir, lo que escupe por salida estándar.



An InputStream reads bytes



A Reader works with an InputStream and reads characters



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

Gestión de la E/S

getInputStream()

```
Process p = pbuilder.start();
BufferedReader processOutput = new BufferedReader(new InputStreamReader(p.getInputStream()));
String linea;
while ((linea = processOutput.readLine()) != null) {
    System.out.println("> " + linea);
}
processOutput.close();
```

Gestión de la E/S

getErrorStream()

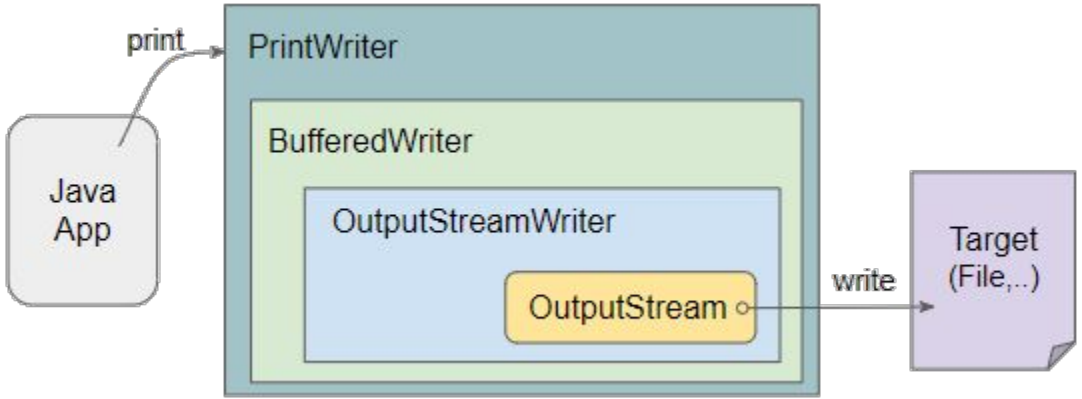
Además de la salida estándar, también podemos obtener la salida de error (stderr) que genera el proceso hijo para procesarla desde el padre.

```
Process p = pbuilder.start();
BufferedReader processErrorOutput = new BufferedReader(new InputStreamReader(p.getErrorStream()));
String linea;
while ((linea = processErrorOutput.readLine()) != null) {
    System.out.println("> " + linea);
}
processErrorOutput.close();
```

Gestión de la E/S

getOutputStream()

También podemos enviar información desde el proceso padre al proceso hijo, usando el último de los tres streams que nos queda, el *stdin*.



Gestión de la E/S

getOutputStream()

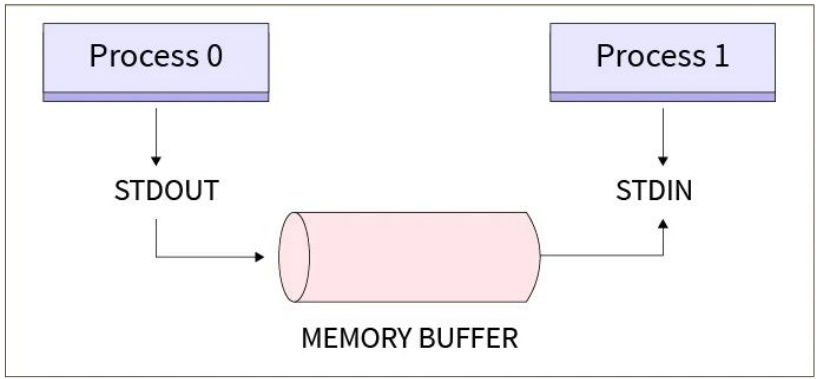
```
PrintWriter toProcess = new PrintWriter(
    new BufferedWriter(
        new OutputStreamWriter(
            p.getOutputStream(), "UTF-8")), true);

toProcess.println("sent to child");
```

Gestión de la E/S

Pipe

En Linux, un pipe (tubería) es una característica que permite conectar la salida de un proceso a la entrada de otro, creando una cadena de procesamiento de datos.

- Un pipe redirige la salida estándar (stdout) de un comando a la entrada estándar (stdin) de otro comando. Esto se realiza utilizando el carácter |
- Cuando usas un pipe, el sistema operativo crea un buffer en memoria que actúa como un canal de comunicación entre los procesos.



Gestión de la E/S

Pipe

En Java, podemos crear un pipe entre dos procesos P1 y P2 conectando la salida estándar de P1 con la entrada estándar de P2.

```
try (InputStream inputStream = process1.getInputStream(); OutputStream outputStream =
process2.getOutputStream()) {
      buffer = new byte[1024];
      int bytesRead;
      while ((bytesRead = inputStream.read(buffer)) != -1) {
            outputStream.write(buffer, 0, bytesRead);
      }
      outputStream.flush();
}
```

Usamos un buffer para leer datos del flujo de salida del primer proceso y escribirlos en el flujo de entrada del segundo proceso.

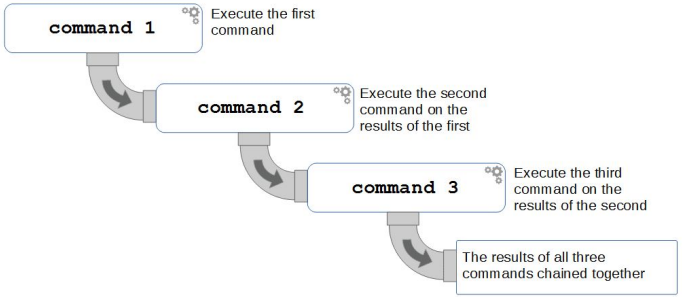
Gestión de la E/S

Pipeline

Este proceso se puede simplificar utilizando el método *startPipeline* del API de *ProcessBuilder.*

```
public static List<Process> startPipeline(List<ProcessBuilder> builders)
```

Este método usa un lista de objetos *ProcessBuilder* y se encarga de lanzar un proceso para cada uno de los *ProcessBuilder* recibidos. Además automatiza la creación de tuberías encadenadas (pipeline) haciendo que la salida de cada proceso esté enlazada con la entrada del siguiente.



Gestión de la E/S

Pipeline

Por ejemplo, si queremos realizar este tipo de operaciones tan comunes en shellscript:

```
find . -name *.java -type f | wc -l
```

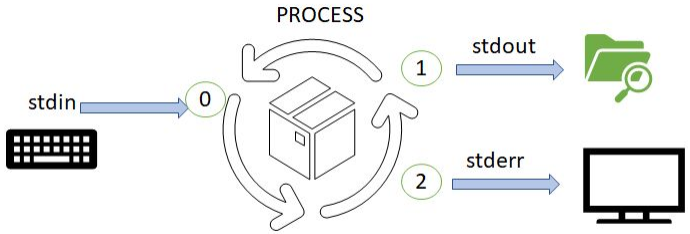
```
List<ProcessBuilder> builders = Arrays.asList(
      new ProcessBuilder("find", "src", "-name", "*.java", "-type", "f"),
      new ProcessBuilder("wc", "-l"));

List<Process> processes = ProcessBuilder.startPipeline(builders);
Process last = processes.get(processes.size() - 1);

// Desde el proceso padre podemos recoger la salida del último proceso para el resultado final del pipeline
```

Gestión de la E/S

Redirección de la Entrada y Salida

En un sistema real, probablemente necesitemos guardar los resultados de un proceso en un archivo de log o de errores para su posterior análisis. La API de *ProcessBuilder* proporciona métodos para hacer exactamente eso.

- `redirectInput(File file)`
- `redirectOutput(File file)`
- `redirectError(File file)`

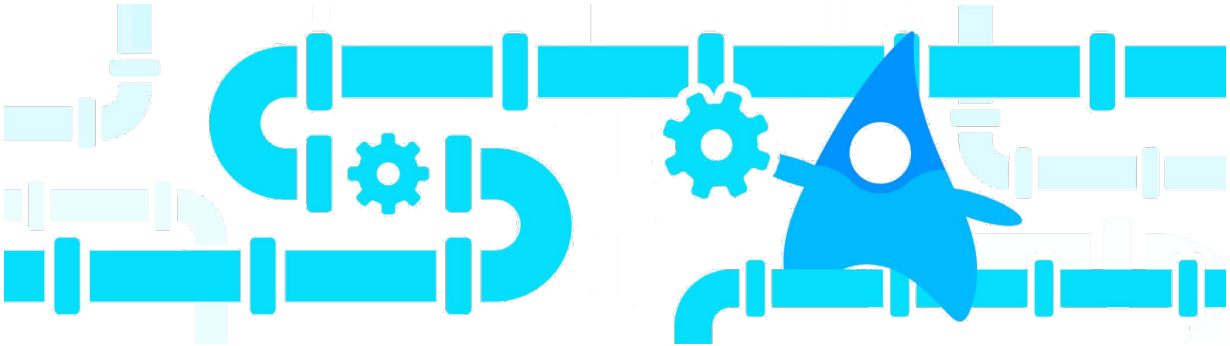


Gestión de la E/S

Redirección de la Entrada y Salida

Por defecto los procesos hijos reciben la entrada a través de una tubería a la que podemos acceder usando el *OutputStream* que nos devuelve *Process.getOutputStream().*

Sin embargo, esa entrada estándar se puede cambiar y redirigirse a otros destinos como un fichero usando el método *redirectOutput(File)*. Si modificamos la salida estándar, el método *getOutputStream()* devolverá *ProcessBuilder.NullOutputStream*.



Gestión de la E/S

Redirección de la Entrada y Salida

Vamos a ver con un ejemplo cómo hacer un programa que muestre la versión de Java. La salida se va a guardar en un archivo de log en vez de enviarla al padre por la tubería de salida estándar:

```
ProcessBuilder processBuilder = new ProcessBuilder("java", "-version");
// La salida de error se enviará al mismo sitio que la estándar
processBuilder.redirectErrorStream(true);
File log = folder.newFile("java-version.log");
processBuilder.redirectOutput(log);
Process process = processBuilder.start();
```

Es lo mismo que si llamáramos a nuestra aplicación usando el operador de redirección de salida:

```
java ejemplo-java-version > java-version.log
```

Gestión de la E/S

Redirección de la Entrada y Salida

Ahora queremos añadir (append to) información al archivo de log file en vez de sobreescribir el archivo cada vez que se ejecuta el proceso.

```
File log = new File("java-version-append.log");
processBuilder.redirectErrorStream(true);
processBuilder.redirectOutput(Redirect.appendTo(log));
```

Gestión de la E/S

Redirección de la Entrada y Salida

Para hacer las redirecciones también podemos utilizar la clase *ProcessBuilder.Redirect* como parámetro para los métodos anteriores, utilizando uno de los siguientes valores:

Valor	Significado
Redirect.DISCARD	La información se descarta
Redirect.to(File)	La información se guardará en el fichero indicado. Si existe, se vacía.
Redirect.from(File)	La información se leerá del fichero indicado
Redirect.appendTo(File)	La información se añadirá en el fichero indicado. Si existe, no se vacía

Información de los procesos en Java

Una vez que un proceso está en ejecución podemos obtener información acerca de ese proceso usando los métodos de la clase *java.lang.ProcessHandle y java.lang.ProcessHandle.Info*:

- El comando usado para lanzar el proceso
- Los argumentos/parámetros que recibió el proceso
- El instante de tiempo en el que se inició el proceso
- El tiempo de CPU que ha usado el proceso y el usuario que lo ha lanzado
- PID del proceso

Información de los procesos en Java

Información del proceso padre

En este ejemplo obtenemos la información del proceso actual (*ProcessHandle.current()*), así que si estamos en el proceso padre, sólo podemos mostrar su información, pero no la de su hijo.

```
ProcessHandle processHandle = ProcessHandle.current();
ProcessHandle.Info processInfo = processHandle.info();
System.out.println("PID: " + processHandle.pid());
System.out.println("Arguments: " + processInfo.arguments());
System.out.println("Command: " + processInfo.command());
System.out.println("Instant: " + processInfo.startInstant());
System.out.println("Total CPU duration: " + processInfo.totalCpuDuration());
System.out.println("User: " + processInfo.user());
```

Información de los procesos en Java

Información del proceso hijo

También es posible acceder a la información de un proceso lanzado (proceso hijo). En este caso necesitamos la instancia de java.lang.Process para llamar a su método *toHandle()* y obtener la instancia de *java.lang.ProcessHandle* del proceso hijo.

```
Process process = processBuilder.start();
ProcessHandle childProcessHandle = process.toHandle();
ProcessHandle.Info child processInfo = childProcessHandle.info();
```