

1. Árbol de procesos

El sistema operativo es el encargado de crear y gestionar los nuevos procesos siguiendo las directrices del usuario.

Así, cuando un usuario quiere abrir un programa, el sistema operativo es el responsable de crear y poner en ejecución el proceso correspondiente que lo ejecutará. Aunque el responsable del proceso de creación es el sistema operativo, ya que es el único que puede acceder a los recursos del ordenador, el nuevo proceso se crea siempre por petición de otro proceso. La puesta en ejecución de un nuevo proceso se produce debido a que hay un proceso en concreto que está pidiendo su creación en su nombre o en nombre del usuario.

En este sentido, cualquier proceso en ejecución siempre depende del proceso que lo creó, estableciéndose un vínculo entre ambos. A su vez, el nuevo proceso puede crear nuevos procesos, formándose lo que se denomina un **árbol de procesos**. Cuando se arranca el ordenador, y se carga en memoria el kernel del sistema a partir de su imagen en disco, se crea el proceso inicial del sistema. A partir de este proceso, se crea el resto de procesos de forma jerárquica, estableciendo padres, hijos, abuelos, etc.

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

PID	Process Name
0	kernel_task
1 ▼	launchd
113	mds
158	WindowServer
138	coreservicesd
48054	diskimages-helper
115	loginwindow
45649	fsevents
301	coreaudiod
55644	cupsd
15 ▼	configd
49045	eapolclient

Árbol de procesos de Mac OS X. Se puede observar cómo el proceso principal del cual cuelgan el resto de procesos es el proceso con PID 0

2. Operaciones básicas con procesos

Siguiendo el vínculo entre procesos establecido en el árbol de procesos, el proceso creador se denomina **padre** y el proceso creado se denomina **hijo**. A su vez, los hijos pueden crear nuevos hijos. A la operación de creación de un nuevo proceso la denominaremos **create**.

Cuando se crea un nuevo proceso tenemos que saber que padre e hijo se ejecutan concurrentemente. Ambos procesos comparten la CPU y se irán intercambiando siguiendo la **política de planificación del sistema operativo** para proporcionar multiprogramación. Si el proceso padre necesita esperar hasta que el hijo termine su ejecución para poder continuar la suya con los resultados obtenidos por el hijo, puede hacerlo mediante la operación **wait**.

Los procesos son independientes y tienen su propio espacio de memoria asignado, llamado **imagen de memoria**. Padres e hijos son procesos y, aunque tengan un vínculo especial, mantienen esta restricción. Ambos usan espacios de memoria independientes. En general, parece que el hijo ejecuta un programa diferente al padre, pero en algunos sistemas operativos esto no tiene por qué ser así. Por ejemplo, mientras que en sistemas tipo Windows existe una función **createProcess()**

que crea un nuevo proceso a partir de un programa distinto al que está en ejecución, en sistemas tipo UNIX, la operación a utilizar es **fork()**, que crea un proceso hijo con un duplicado del espacio de direcciones del padre, es decir, un duplicado del programa que se ejecuta desde la misma posición. Sin embargo, en ambos casos, los padres e hijos (aunque sean un duplicado en el momento de la creación en sistemas tipos UNIX) son independientes y las modificaciones que uno haga en su espacio de memoria, como escritura de variables, no afectarán al otro.

Como padre e hijo tienen espacios de memoria independientes, pueden compartir recursos para intercambiarse información. Estos recursos pueden ir desde ficheros abiertos hasta zonas de memoria compartida. La memoria compartida es una región de memoria a la que pueden acceder varios procesos cooperativos para compartir información. Los procesos se comunican escribiendo y leyendo datos en dicha región. El sistema operativo solamente interviene a la hora de crear y establecer los permisos de qué procesos pueden acceder a dicha zona. Los procesos son los responsables del formato de los datos compartidos y de su ubicación.

Al terminar la ejecución de un proceso, es necesario avisar al sistema operativo de su terminación para que de esta forma el sistema libere, si es posible, los recursos que tenga asignados. En general, es el propio proceso el que le indica al sistema operativo mediante una operación denominada **exit** que quiere terminar, pudiendo aprovechar para mandar información respecto a su finalización al proceso padre en ese momento.

El proceso hijo depende tanto del sistema operativo como del proceso padre que lo creó. Así, el padre puede terminar la ejecución de un proceso hijo cuando crea conveniente. Entre estos motivos podría darse que el hijo excediera el uso de algunos recursos o que la funcionalidad asignada al hijo ya no sea necesaria por algún motivo.

Para ello puede utilizar la operación **destroy**. Esta relación de dependencia entre padre e hijo, lleva a casos como que si el padre termina, en algunos sistemas operativos no se permita que sus hijos continúen la ejecución, produciéndose lo que se denomina “**terminación en cascada**”.

En definitiva, cada sistema operativo tiene unas características únicas, y la gestión de los procesos es diferente en cada uno de ellos. Por simplificación y portabilidad, evitando así depender del sistema operativo sobre el cual se esté ejecutando vamos a explicar la gestión de procesos para la máquina virtual de **Java (Java Virtual Machine [JVM])**. JVM es un entorno de ejecución ligero y gratuito multiplataforma que permite la ejecución de binarios o bytecode del lenguaje de programación Java sobre cualquier sistema operativo, salvando las diferencias entre ellos.

Respecto a la utilización de **Java**, tenemos que saber que los procesos padre e hijo en la JVM no tienen por qué ejecutarse de forma concurrente. Además, no se produce terminación en cascada, pudiendo sobrevivir los hijos a su padre ejecutándose de forma asíncrona. Por último, hay que tener en cuenta que puede no funcionar bien para procesos especiales en ciertas plataformas nativas (por ejemplo, utilización de ventanas nativas en MS-DOS/Windows, shell scripts en GNU Linux/UNIX/Mac OS, etc.).

3. Creación de procesos en Java

Las clases que se usan en Java para la creación de procesos, están dentro del paquete `java.lang` y son:

- **Clase `java.lang.Process`.** Proporciona los objetos Proceso, por los que podremos controlar los procesos creados desde nuestro código.
- **Clase `java.lang.Runtime`.** Clase que permite lanzar la ejecución de un programa en el sistema.

Los métodos más interesantes de la clases `Runtime` son:

- **`Process exec(String comando)`:** devuelve un objeto **`Process`** que representa al proceso en ejecución que está realizando la tarea comando. La ejecución del método **`exec()`** puede lanzar las excepciones: **`SecurityException`**, si hay administración de seguridad y no tenemos permitido crear subprocesos. **`IOException`**, si ocurre un error de E/S.

NullPointerException y **IllegalArgumentException**, si comando es una cadena nula o vacía.

- **static Runtime getRuntime():** devuelve el objeto Runtime asociado con la aplicación Java en curso.

Ejemplo que muestra cómo se puede ejecutar una aplicación de Windows, en este caso el bloc de notas.

```
1 public class EjemploNotepad{
2     public static void main(String[] args){
3         Runtime r=Runtime.getRuntime();
4         String comando="NOTEPAD";
5         Process p;
6         try{
7             p=r.exec(comando);
8         }catch(Exception e){
9             System.out.println("Error en: "+comando);
10            e.printStackTrace();
11        }
12    }
13 }
14 }
15 }
```

Lo compilamos y lo ejecutamos . Al ejecutarlo debería abrirse el bloc de notas de Windows.

- javac EjemploNotepad.java
- java EjemploNotepad

Si estamos trabajando en Linux, no abriremos Notepad, sino Gedit.

```
public class EjemploGedit{
    public static void main(String[] args){
        Runtime r=Runtime.getRuntime();
        String comando="gedit";
        Process p;
        try{
            p=r.exec(comando);
        }catch(Exception e){
            System.out.println("Error en: "+comando);
            e.printStackTrace();
        }
    }
}
```

Para los comandos de Window que no tienen ejecutable (como DIR) es necesario utilizar el comando CMD.EXE. Entonces para hacer DIR desde un program Java tendríamos que escribir:

String comando="CMD /C DIR";

CMD inicia una nueva instancia del intérprete de comandos de Windows. Para ver la sintaxis del comando escribimos desde el indicador de DOS: HELP CMD.

Para ejecutar un comando escribimos:

- CMD /C comando: ejecuta el comando especificado y finaliza.
- CMD /K comando: ejecuta el comando especificado pero sigue activo.

En el ejemplo anterior no obtendremos ninguna salida probando el comando CMD /C DIR, ya que la salida del comando se dirige a nuestro programa Java, no al pantalla.

Para leer la salida, es decir, lo que nos devuelve el método *exec()* del **Runtime**, tenemos que usar el objeto **Process**, que se obtenía así en el ejemplo:

```
p=r.exec(comando);
```

La clase Process posee el método *getInputStream()* que nos permite leer el stream de salida del proceso, es decir, podemos leer lo que el comando que ejecutamos escribió en la consola. Definiremos así el stream:

```
Process p = Runtime.getRuntime().exec("CMD /C DIR");
InputStream is= p.getInputStream();
BufferedReader br = new BufferedReader(new InputStreamReader(is));
```

Para leer la salida usamos el método *readLine()* de **BufferedReader** que nos devuelve una línea de texto. A continuación se muestra la ejecución del comando DIR de Windows desde un programa Java.

```
import java.io.*;

public class EjemploDIR{

    public static void main(String[] args){
        Runtime r=Runtime.getRuntime();
        String comando = "CMD /C DIR";
        Process p=null;
        try{
            p = r.exec(comando);
            InputStream is= p.getInputStream();
            BufferedReader br = new BufferedReader(new InputStreamReader(is));
            String linea;
            while((linea=br.readLine())!=null)    //lee una linea
                System.out.println(linea);
            br.close();
        }
        catch(Exception e){
            e.printStackTrace();
        }

        //Comprobación de error - o bien -1 mal
        int exitVal;
        try{
            exitVal=p.waitFor();
            System.out.println("Valor de salida: "+ exitVal);
        }
        catch(InterruptedException e){
            e.printStackTrace();
        }
    }
}
```

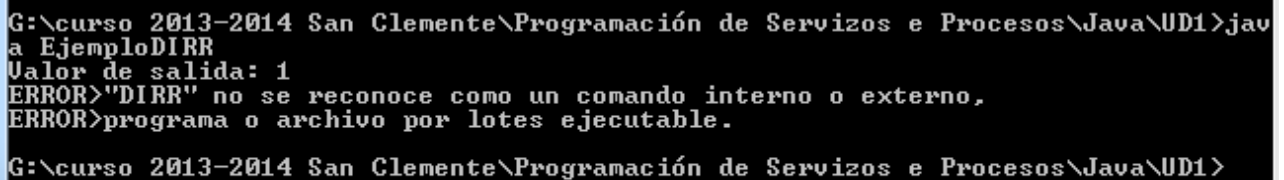
El método ***waitFor()*** hace que el proceso actual espere hasta que el subprocesso representado por el objeto **Process** finalice. Devuelve 0 si ha finalizado correctamente.

La clase **Process** posee el método ***getErrorStream()*** que nos va permitir obtener un stream para poder leer los posibles errores que se produzcan al lanzar el proceso. En el ejemplo anterior si cambiamos el comando y escribimos algo incorrecto, por ejemplo “CMD /C DIRR” al ejecutarlo aparecerá como valor de salida 1 indicando que el proceso no ha finalizado correctamente.

Si añadimos el siguiente código al ejemplo:

```
try{
    InputStream er = p.getErrorStream();
    BufferedReader brer = new BufferedReader(new InputStreamReader(er));
    String liner=null;
    while((liner=brer.readLine())!=null) //lee una linea
        System.out.println("ERROR>" + liner);
    brer.close();
}
catch(Exception e){
    e.printStackTrace();
}
}
```

Se obtendrá la siguiente salida indicando el error que se ha producido:



```
G:\curso 2013-2014 San Clemente\Programación de Servicios e Procesos\Java\UD1>jav
a EjemploDIRR
Valor de salida: 1
ERROR>"DIRR" no se reconoce como un comando interno o externo,
ERROR>programa o archivo por lotes ejecutable.
G:\curso 2013-2014 San Clemente\Programación de Servicios e Procesos\Java\UD1>
```

De todos modos, tenemos que tener en cuenta que el método ***exec()*** no actúa como un intérprete de comandos o Shell, lo que hace es ejecutar un programa, pero no es una línea de comandos. Si por ejemplo quisiésemos hacer que la salida de un programa vaya a un fichero tendríamos que hacerlo mediante programación.

Si estamos trabajando en Linux, en lugar de llamar al CMD, escribimos el comando directamente, de esta manera:

```
String comando="ls -l";
```

```

import java.io.*;
public class EjemploLS{
    public static void main(String[] args){
        Runtime r=Runtime.getRuntime();
        String comando = "ls -l";
        Process p=null;
        try{
            p = r.exec(comando);
            InputStream is= p.getInputStream();
            BufferedReader br = new BufferedReader(new InputStreamReader(is));
            String linea;
            while((linea=br.readLine())!=null)    //lee una linea
                System.out.println(linea);
            br.close();
        }
        catch(Exception e){
            e.printStackTrace();
        }

        //Comprobacion de error - o bien -1 mal
        int exitVal;
        try{
            exitVal=p.waitFor();
            System.out.println("Valor de salida: "+ exitVal);
        }
        catch(InterruptedException e){
            e.printStackTrace();
        }
    }
}

```

La clase ProcessBuilder

La versión 1.5 de JDK (y posteriores) añade una nueva forma de creación y ejecución de procesos del sistema operativo mediante la clase **ProcessBuilder**. Igual que **Process** y **Runtime** pertenece al paquete *java.lang*. Cada instancia **ProcessBuilder** gestiona una colección de atributos de proceso. El método **start()** crea una nueva instancia de **Process** con esos atributos y puede ser invocado varias veces desde la misma instancia para crear nuevos subprocesos con atributos idénticos o relacionados.

Los métodos **ProcessBuilder.start()** y **Runtime.exec()** crean un proceso nativo en el sistema operativo subyacente donde se está ejecutando la JVM y devuelven un objeto Java de la clase **Process** que puede ser utilizado para controlar dicho proceso.

- **Process ProcessBuilder.start():** inicia un nuevo proceso utilizando los atributos indicados en el objeto. El nuevo proceso ejecuta el comando y los argumentos indicados en el método **command()**, ejecutándose en el directorio de trabajo especificado por **directory()**, utilizando las variables de entorno definidas en **environment()**.
- **Process Runtime.exec(String[] cmdarray, String[] envp, File dir):** ejecuta el comando especificado y argumentos en **cmdarray** en un proceso hijo independiente con el entorno **envp** (variables de entorno y sus valores) y el directorio de trabajo especificado en **dir**.

Ambos métodos comprueban que el comando a ejecutar es un comando o ejecutable válido en el sistema operativo subyacente sobre el que ejecuta la JVM. El ejecutable se ha podido obtener mediante la compilación de código en cualquier lenguaje de programación. Al final, crear un nuevo proceso depende del sistema operativo en concreto que esté ejecutando por debajo de la JVM. En este sentido, pueden ocurrir múltiples problemas, como:

- No encuentra el ejecutable debido a la ruta indicada.
- No tener permisos de ejecución.
- No ser un ejecutable válido en el sistema.
- etc.

En la mayoría de los casos, se lanza una excepción dependiente del sistema en concreto, pero siempre será una subclase de `IOException`.

Para iniciar un nuevo proceso que utiliza el directorio de trabajo y el entorno del proceso en curso escribimos la siguiente orden:

```
Process p = new ProcessBuilder("Comando", "Argum1").start();
```

Para usar **ProcessBuilder** en los ejemplos anteriores no es necesario usar el método *exec()* de **Runtime**. Por ejemplo, en el ejemplo de listar el directorio construimos y ejecutamos la orden de la siguiente manera:

```
ProcessBuilder pb=new ProcessBuilder("CMD", "/C", "DIR");
Process p = pb.start();
```

Ejemplo de creación de un proceso utilizando ProcessBuilder:

```
import java.io.IOException;
import java.util.Arrays;

public class RunProcess {
    public static void main(String[] args) throws IOException {
        if (args.length <= 0) {
            System.err.println("Se necesita un programa a ejecutar");
            System.exit(-1);
        }
        ProcessBuilder pb = new ProcessBuilder(args);
        try {
            Process process = pb.start();
            int retorno = process.waitFor();
            System.out.println("La ejecucion de " +
                Arrays.toString(args) + " devuelve " + retorno);
        } catch (IOException ex) {
            System.err.println("Excepcion de E/S!!");
            System.exit(-1);
        } catch (InterruptedException ex) {
            System.err.println("El proceso hijo finalizo de forma incorrecta");
            System.exit(-1);
        }
    }
}
```

Para ejecutarlo indicariamos, separados por espacios, el comando y cada uno de sus argumentos:

```
L:\psp1>java RunProcess cmd /c dir
La ejecucion de [cmd, /c, dir] devuelve 0
```

En el siguiente ejemplo vamos ver el uso de varios métodos de la clase **ProcessBuilder**: *environment()* que devuelve las variables de entorno del proceso; el método *command()* sin parámetros, que devuelve los argumentos del proceso definido en *test*; y con parámetros donde se define un nuevo proceso y sus argumentos. Después se ejecutará este último proceso.


```

import java.io.*;
import java.util.*;

public class EjemploPB1{
    public static void main(String args[]){
        ProcessBuilder test= new ProcessBuilder();
        Map entorno= test.environment();
        System.out.println("Variables de entorno:");
        System.out.println(entorno);

        test = new ProcessBuilder("java", "RunProcess", "EjemploNotepad");

        //devuelve el nombre del proceso y sus argumentos.
        List l = test.command();
        Iterator iter = l.iterator();
        System.out.println("Argumentos del comando:");
        while(iter.hasNext())
            System.out.println(iter.next());

        //ejecutamos el comando DIR.
        test=test.command("CMD", "/C", "DIR");
        try{
            Process p= test.start();
            InputStream is = p.getInputStream();
            BufferedReader br = new BufferedReader(new InputStreamReader(is));
            String linea;
            while((linea=br.readLine())!=null)    //lee una linea
                System.out.println(linea);
            br.close();
        }

        catch(Exception e){
            e.printStackTrace();
        }
    }
}

```

La compilación y la ejecución muestra la siguiente salida:

```

C:\Windows\system32\cmd.exe
F:\curso 2013-2014 San Clemente\Programación de Servicios e Procesos\Java\UD1>jav
ac EjemploPB1.java

F:\curso 2013-2014 San Clemente\Programación de Servicios e Procesos\Java\UD1>jav
a EjemploPB1
Variables de entorno:
<ProgramData=C:\ProgramData, USERPROFILE=C:\Users\agnovoa, PATHEXT=.COM;.EXE;.BA
T;.CMD;.UBS;.UBE;.JS;.JSE;.WSF;.WSH;.MSC, USERDNSDOMAIN=SANCLEMENTE.LOCAL, windo
ws_tracing_logfile=C:\BUTBin\Tests\installpackage\csilogfile.log, ProgramFiles(x
86)=C:\Program Files (x86), =ExitCode=00000000, SystemDrive=C:, TEMP=C:\Users\ag
novoa\AppData\Local\Temp, windows_tracing_flags=3, ProgramFiles=C:\Program Files
, Path=C:\Program Files\Java\jdk1.7.0_07\bin;C:\Program Files\Microsoft SQL Serv
er\100\Binn\;C:\Program Files\Intel\DMIX;C:\Program Files\Java\jdk1.7.0_03\h
in;C:\Program Files\Java\jdk1.7.0_02\bin;C:\Windows\system32;C:\Windows;C:\Wind
ows\System32\Wbem;C:\Windows\System32\WindowsPowerShell\v1.0\;C:\Program Files (x
86)\WinSCP;C:\Program Files (x86)\Nmap;C:\ProgramData\Microsoft\Windows\Start M
enu\Programs\Control Remoto - FTP\Putty;C:\Program Files (x86)\Microsoft SQL Ser
ver\100\Tools\Binn\Usshell\Common7\IDE;C:\Program Files (x86)\Microsoft SQL Ser
ver\100\Tools\Binn\;C:\Program Files\Microsoft SQL Server\100\Tools\Binn\;C:\Pro
gram Files (x86)\Microsoft SQL Server\100\Binn\;C:\Program Files (x86)\Quick
Time\QTSystem\;C:\Program Files\android-sdk-windows\tools;C:\Program Files\andro
id-sdk-windows\platform-tools, HOMEDRIVE=L:, PROCESSOR_REVISION=170a, =C:=C:\, U
SERDOMAIN=SANCLEMENTE, QTJAVA=C:\Program Files (x86)\Java\jre7\lib\ext\QTJava.zi
p, ALLUSERSPROFILE=C:\ProgramData, UBOX_INSTALL_PATH=C:\Program Files\Oracle\Vir
tualBox\, ProgramW6432=C:\Program Files, PROCESSOR_IDENTIFIER=Intel64 Family 6 M
odel 23 Stepping 10, GenuineIntel, SESSIONNAME=Console, TMP=C:\Users\agnovoa\AppData\Local\Temp, PROCESSOR_ARCHITECTURE=AMD64, LOGONSERVER=\\PEDROSO, =:::.\, C
LASSPATH=.;C:\Program Files (x86)\Java\jre7\lib\ext\QTJava.zip;C:\Program Files\
Java\conectores\mysql-connector.zip, CommonProgramFiles=C:\Program Files\Common
Files, OS=Windows_NT, FP_NO_HOST_CHECK=NO, HOMEPATH=\\, PROMPT=$P$G, US100COMNT00
LS=C:\Program Files (x86)\Microsoft Visual Studio 10.0\Common7\Tools\, PROCESSOR
_LEVEL=6, CommonProgramW6432=C:\Program Files\Common Files, LOCALAPPDATA=C:\User
s\agnovoa\AppData\Local, HOMESHARE=\\White\Profes$agnovoa, COMPUTERNAME=INFO02,
SystemRoot=C:\Windows, windir=C:\Windows, NUMBER_OF_PROCESSORS=2, PSModulePath=
C:\Windows\system32\WindowsPowerShell\v1.0\Modules\, PUBLIC=C:\Users\Public, USE
RNAME=agnovoa, CommonProgramFiles(x86)=C:\Program Files (x86)\Common Files, ComS
pec=C:\Windows\system32\cmd.exe, =F=F:\curso 2013-2014 San Clemente\Programaci
n de Servicios e Procesos\Java\UD1, APPDATA=C:\Users\agnovoa\AppData\Roaming>
Argumentos del comando:
java
RunProcess
EjemploNotepad
El volumen de la unidad F es Ana WD 1Th
El número de serie del volumen es: DC89-3951

Directorio de F:\curso 2013-2014 San Clemente\Programación de Servicios e Proces
os\Java\UD1

25/09/2013 11:50 <DIR> .
25/09/2013 11:50 <DIR> ..
24/09/2013 18:07 1.485 EjemploDIR.class
24/09/2013 18:10 744 EjemploDIR.java
24/09/2013 18:46 1.706 EjemploDIRR.class
24/09/2013 18:46 1.089 EjemploDIRR.java
24/09/2013 12:45 941 EjemploNotepad.class
24/09/2013 12:39 273 EjemploNotepad.java
25/09/2013 11:50 1.749 EjemploPB1.class
25/09/2013 11:49 1.019 EjemploPB1.java
24/09/2013 18:18 1.498 LanzaComando.class
24/09/2013 18:18 755 LanzaComando.java
24/09/2013 19:45 1.324 RunProcess.class
24/09/2013 10:43 753 RunProcess.java
12 archivos 13.336 bytes
2 dirs 665.190.772.736 bytes libres

F:\curso 2013-2014 San Clemente\Programación de Servicios e Procesos\Java\UD1>

```

La versión 1.7 del JDK y posteriores proporciona métodos que nos permiten redirigir la salida estándar y de error a otro fichero. Se trata de los métodos ***redirectOutput()*** y ***redirectError()***. El siguiente ejemplo ejecuta el comando “CMD /C DIR” y envía la salida al fichero *salida.txt*, si ocurre algún error se envía a *error.txt*:

```
import java.io.File;
import java.io.IOException;

public class EjemploPB2{
    public static void main(String args[]){
        ProcessBuilder pb= new ProcessBuilder("CMD", "/C", "DIR");

        File fOut = new File("E:\\java\\UD1\\salida.txt");
        File fErr = new File("E:\\java\\UD1\\error.txt");

        pb.redirectOutput(fOut);
        pb.redirectError(fErr);
        try{
            pb.start();
        }
        catch(IOException ex){
            System.err.println("Excepcion de E/S!!");
            System.exit(-1);
        }
    }
}
```

Debemos recordar que trabajando con Java en Windows, se separan los directorios con doble barra inclinada (\\).

4. Terminación de procesos en Java

Un proceso puede terminar de forma abrupta un proceso hijo que creó. Para ello el proceso padre puede ejecutar la operación ***destroy***. Esta operación elimina el proceso hijo indicado liberando sus recursos en el sistema operativo subyacente. En caso de Java, los recursos correspondientes los eliminará el ***garbage collector*** cuando considere.

Si no se fuerza la finalización de la ejecución del proceso hijo de forma anómala, el proceso hijo realizará su ejecución completa terminando y liberando sus recursos al finalizar.

En el ejemplo se ve como se crea un proceso mediante **Runtime** para después destruirlo.

```
import java.io.IOException;
import java.util.Arrays;

public class RuntimeProcess {
    public static void main(String[] args) {
        if (args.length <= 0) {
            System.err.println("Se necesita un programa a ejecutar");
            System.exit(-1);
        }

        Runtime runtime = Runtime.getRuntime();
        try {
            Process process = runtime.exec(args);
            process.destroy();

            int retorno = process.waitFor();
            System.out.println("La ejecucion de " +
                Arrays.toString(args) + " devuelve " + retorno);
        } catch (IOException ex) {
            System.err.println("Excepción de E/S!!");
            System.exit(-1);
        } catch (InterruptedException ex) {
            System.err.println("Excepción Interrupción");
            System.exit(-1);
        }
    }
}
```

5. Comunicación de procesos en Java

Es importante recordar que 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. No se refiere a los parámetros de ejecución del programa. Normalmente suele ser el teclado, pero podría recibirlos de un fichero, de la tarjeta de red o hasta de otro proceso, entre otros sitios. La lectura de datos a lo largo de un programa leerá los datos de su entrada estándar.
- **La salida estándar (stdout):** sitio donde el proceso escribe los resultados que obtiene. Normalmente es la pantalla, aunque podría ser, entre otros, la impresora o hasta otro proceso que necesite esos datos como entrada. La escritura de datos que se realice en un programa (por ejemplo mediante `System.out.println` en Java) se produce por la salida estándar.
- **La salida de error (stderr):** sitio donde el proceso envía los mensajes de error. Habitualmente es el mismo que la salida estándar, pero puede especificarse que sea otro lugar, por ejemplo un fichero para identificar más fácilmente los errores que ocurren 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 la mayoría de los sistemas operativos, estas entradas y salidas en proceso hijo son una copia de las mismas entradas y salidas que tuviera su padre. De tal forma que si un proceso lee de un fichero y muestra la salida estándar por pantalla, su hijo correspondiente leerá del mismo fichero y escribirá en pantalla. En Java, en cambio, el proceso hijo creado de la clase **Process** no tiene su propia interfaz de comunicación, por lo que el usuario no puede comunicarse con él directamente. Todas sus salidas y entradas de información (stdin, stdout y stderr) se redirigen al proceso padre a través de los siguientes flujos de datos o streams:

- **OutputStream:** flujo de salida del proceso hijo. El stream está conectado por un pipe a la entrada estándar (stdin) del proceso hijo.
- **InputStream:** flujo de entrada del proceso hijo. El stream está conectado por un pipe a la salida estándar (stdout) del proceso hijo.
- **ErrorStream:** flujo de error del proceso hijo. El stream está conectado por un pipe a la salida estándar (stderr) del proceso hijo. Sin embargo, hay que saber que, por defecto, para la JVM, stderr está conectado al mismo sitio que stdout.

Si se desea tenerlos separados, lo que permite identificar errores de forma más sencilla, se puede utilizar el método `redirectErrorStream(boolean)` de la clase `ProcessBuilder`. Si se pasa un valor `true` como parámetro, los flujos de datos correspondientes a stderr y stdout en la JVM serán diferentes y representarán la salida estándar y la salida de error del proceso de forma correspondiente.

Utilizando estos streams, el proceso padre puede comunicarse con el proceso hijo enviándole datos y recibir los resultados de salida que este genere comprobando los errores.

Hay que tener en cuenta que en algunos sistemas operativos, el tamaño de los buffers de entrada y salida que corresponde a stdin y stdout está limitado. En este sentido, un fallo al leer o escribir en los flujos de entrada o salida del proceso hijo puede provocar que el proceso hijo se bloquee. Por eso, en Java se suele realizar la comunicación padre-hijo a través de un buffer utilizando los streams vistos.

```

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.util.Arrays;
public class CommunicationBetweenProcess {
    public static void main(String args[]) throws IOException {
        Process process = new ProcessBuilder(args).start();
        InputStream is = process.getInputStream();
        InputStreamReader isr = new InputStreamReader(is);
        BufferedReader br = new BufferedReader(isr);
        String line;
        System.out.println("Salida del proceso " +
            Arrays.toString(args) + ":");

        while ((line = br.readLine()) != null) {
            System.out.println(line);
        }
    }
}

```

Para ejecutarlo, habrá que indicarle el programa a ejecutar y sus argumentos, por ejemplo, si ejecutamos el programa “java EjemploDIR” obtenemos la siguiente salida:

```

L:\psp1>java CommunicationBetweenProcess java EjemploDIR
Salida del proceso [java, EjemploDIR]:
El volumen de la unidad L no tiene etiqueta.
El número de serie del volumen es: 4901-AEDB

Directorio de L:\psp1
29/10/2014 13:09 <DIR> .
28/10/2014 12:17 <DIR> ..
24/09/2013 11:39 273 EjemploNotepad.java
28/10/2014 10:47 941 EjemploNotepad.class
24/09/2013 17:10 744 EjemploDIR.java
28/10/2014 10:54 1.485 EjemploDIR.class
24/09/2013 17:46 1.089 EjemploDIRR.java
28/10/2014 10:55 1.706 EjemploDIRR.class
24/09/2013 09:43 753 RunProcess.java
29/10/2014 12:36 1.324 RunProcess.class
25/09/2013 10:49 1.019 EjemploPB1.java
29/10/2014 12:40 1.749 EjemploPB1.class
29/10/2014 12:43 505 EjemploPB2.java
29/10/2014 12:43 982 EjemploPB2.class
29/10/2014 12:43 1.048 salida.txt
29/10/2014 12:43 0 error.txt
21/11/2013 10:53 718 RuntimeProcess.java
29/10/2014 12:48 1.362 RuntimeProcess.class
25/09/2013 11:26 641 CommunicationBetweenProcess.java
29/10/2014 13:09 1.294 CommunicationBetweenProcess.class
18 archivos 17.633 bytes
2 dirs 1.653.260.288 bytes libres
Valor de salida: 0

```

Además de la posibilidad de comunicarse mediante flujos de datos, existen otras alternativas para la comunicación de procesos:

- Usando **sockets** para la comunicación entre procesos.
- Utilizando **JNI (*Java Native Interface*)**. La utilización de Java permite abstraerse del comportamiento final de los procesos en los diferentes sistemas operativos. Se puede utilizar JNI para acceder desde Java a aplicaciones desarrolladas en otros lenguajes de programación de más bajo nivel, como C, que pueden sacar partido al sistema operativo subyacente. A menudo se denomina a JNI como la "válvula de escape" para desarrolladores dado que les permite añadir funcionalidades a sus aplicaciones que el API de Java no puede proporcionar.
- **Librerías de comunicación no estándares** entre procesos en Java que permiten aumentar las capacidades del estándar Java para comunicarlos. Por ejemplo, CLIPC (<http://clipc.sourceforge.net/>) es una librería Java de código abierto que ofrece la posibilidad de utilizar los siguientes mecanismos que no están incluidos directamente en el lenguaje gracias a que utiliza por debajo llamadas a JNI para poder utilizar métodos más cercanos al sistema operativo:

6. Sincronización de procesos en Java**

**Este punto lo veremos más adelante con más detalle

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 *wait*.

- 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 especifica mediante un número entero, cómo resultó 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.

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

Ejemplo:


```

import java.io.IOException;
import java.util.Arrays;
public class ProcessSincronization {
    public static void main(String[] args)
        throws IOException, InterruptedException{
    try{
        Process process = new ProcessBuilder(args).start();
        int retorno = process.waitFor();
        System.out.println("Comando " + Arrays.toString(args) + "devolvió: " + retorno);
    }catch(IOException e){
        System.out.println("Error ocurrió ejecutando el comando:" + e.getMessage());
    }
    catch(InterruptedException e){
        System.out.println("El comando fue interrumpido. Descripción del error: " +
            e.getMessage());
    }
    }
}

```

Lo ejecuto, por ejemplo, para el comando Notepad.

```

L:\psp1>java ProcessSincronization Notepad
Comando [Notepad]devolvió: 0

```

Si quiero probar el método `exitValue()` añado la línea

`int salida=process.exitValue();`

antes del `waitFor()` y si el proceso todavía no ha terminado, debería devolverme una *IllegalThreadStateException*


```

import java.io.IOException;
import java.util.Arrays;
public class ProcessSincronization {
    public static void main(String[] args)
        throws IOException, InterruptedException{
    try{
        Process process = new ProcessBuilder(args).start();
        int salida=process.exitValue();
        int retorno = process.waitFor();
        System.out.println("Comando " + Arrays.toString(args) + "devolvió: " + retorno);
    }catch(IOException e){
        System.out.println("Error ocurrió ejecutando el comando:" + e.getMessage());
    }
    catch(InterruptedException e){
        System.out.println("El comando fue interrumpido. Descripción del error: " +
            e.getMessage());
    }
    }
}

```

Si ejecuto Java ProcessSincronization java EjemploPB2 saltará la excepción, ya que el proceso aún no ha terminado.

```

C:\> Símbolo del sistema

Y:\Java\UD1>java ProcessSincronization java EjemploPB2
Exception in thread "main" java.lang.IllegalThreadStateException: process has not
e
t exited
    at java.lang.ProcessImpl.exitValue(Unknown Source)
    at ProcessSincronization.main(ProcessSincronization.java:8)
Y:\Java\UD1>

```