

# **CAPÍTULO 1: PROGRAMACIÓN DE PROCESOS**

Programación de Servicios y Procesos

# ÍNDICE

- ▶ Concepto básicos
- ▶ Programación concurrente
- ▶ Funcionamiento básico del Sistema Operativo
- ▶ Estados de un proceso
- ▶ Planificación de procesos
- ▶ Gestión de procesos
- ▶ Comunicación de procesos
- ▶ Sincronización de procesos
- ▶ Programación multiproceso

# Conceptos básicos

## ▶ PROGRAMA

- Toda la información (tanto código como datos) almacenada en disco de una aplicación que resuelve una necesidad concreta para los usuarios.

## ▶ PROCESO

- Programa en ejecución. Este concepto no se refiere únicamente al código y a los datos, sino que incluye todo lo necesario para su ejecución:
  - Contador de programa.
  - Imagen de memoria.
  - Estado del procesador.

## ▶ EJECUTABLE

- Fichero que contiene la información necesaria para crear un proceso a partir de los datos almacenados de un programa.

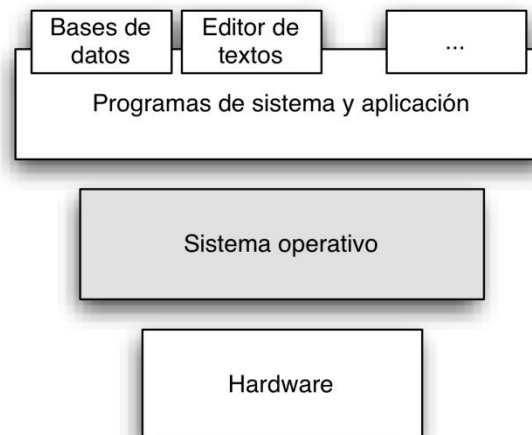
# Conceptos básicos

## ▶ DEMONIO

- Proceso no interactivo que está ejecutándose continuamente en segundo plano. Suele proporcionar un servicio básico para el resto de procesos.

## ▶ SISTEMA OPERATIVO

- Programa que hace de intermediario entre el usuario y las aplicaciones que utiliza y el hardware del ordenador. Entre su funcionalidad.
  - Hace de interfaz entre el usuario y los recursos del ordenador.
  - Permite utilizar los recursos del computador de forma eficiente.
  - Ejecuta los programas del usuario.



# Programación concurrente

- ▶ Actualmente se pueden tener en ejecución al mismo tiempo múltiples tareas interactivas en:
  - Un único procesador (multiprogramación).
    - Si solamente existe un único procesador, solamente un proceso puede estar en un momento determinado en ejecución.
    - El sistema operativo se encarga de cambiar el proceso en ejecución después de un período corto de tiempo (del orden de milisegundos) creando en el usuario la percepción de que múltiples programas se están ejecutando al mismo tiempo (**programación concurrente**).
    - La programación concurrente no mejora el tiempo de ejecución global de los programas ya que se ejecutan intercambiando unos por otros en el procesador. Sin embargo, permite que varios programas parezca que se ejecuten al mismo tiempo.

# Programación concurrente

- Varios núcleos en un mismo procesador (multitarea).
  - Cada núcleo podría estar ejecutando una instrucción diferente al mismo tiempo.
  - El sistema operativo se encarga de planificar los trabajos que se ejecutan en cada núcleo y cambiar unos por otros para generar multitarea.
  - Todos los *cores* comparten la misma memoria por lo que es posible utilizarlos de forma coordinada (**programación paralela**).
  - La programación paralela permite mejorar el rendimiento de un programa ya que permite que se ejecuten varias instrucciones a la vez.

# Programación concurrente

- Varios ordenadores distribuidos en red.
  - Cada uno de los ordenadores tendrá sus propios procesadores y su propia memoria (**programación distribuida**).
  - La programación distribuida posibilita la utilización de un gran número de dispositivos de forma paralela, lo que permite alcanzar elevadas mejoras en el rendimiento de la ejecución de programas distribuidos.
  - Como cada ordenador posee su propia memoria, imposibilita que los procesos puedan comunicarse fácilmente, teniendo que utilizar otros esquemas de comunicación más complejos y costosos a través de la red que los interconecte.

# Funcionamiento básico del SO

## ► KERNEL

- Parte central del sistema operativo responsable de gestionar los recursos del ordenador, permitiendo su uso a través de llamadas al sistema.
  - Parte pequeña del sistema operativo, si la comparamos con lo necesario para implementar su interfaz.
  - A todo lo demás del sistema operativo se le denomina **programas del sistema**.
- Funciona en base a interrupciones. Una **interrupción** es una suspensión temporal de la ejecución de un proceso, para pasar a ejecutar una rutina que trate dicha interrupción.
  1. Cuando salta una interrupción se transfiere el control a la rutina de tratamiento de la interrupción.
  2. Mientras se está atendiendo una interrupción, se deshabilita la llegada de nuevas interrupciones.
  3. Cuando finaliza la rutina, se reanuda la ejecución del proceso en el mismo lugar donde se quedó cuando fue interrumpido.



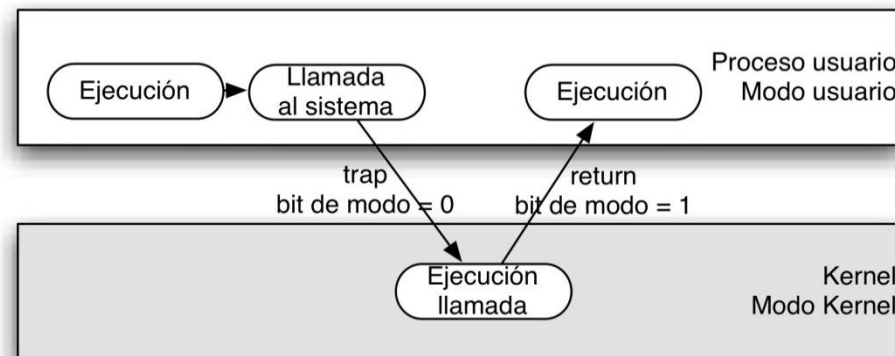
# Funcionamiento básico del SO

## ▶ LLAMADAS AL SISTEMA

- Interfaz que proporciona el *kernel* para que los programas de usuario puedan hacer uso de forma segura de determinadas partes del sistema.

## ▶ MODO DUAL

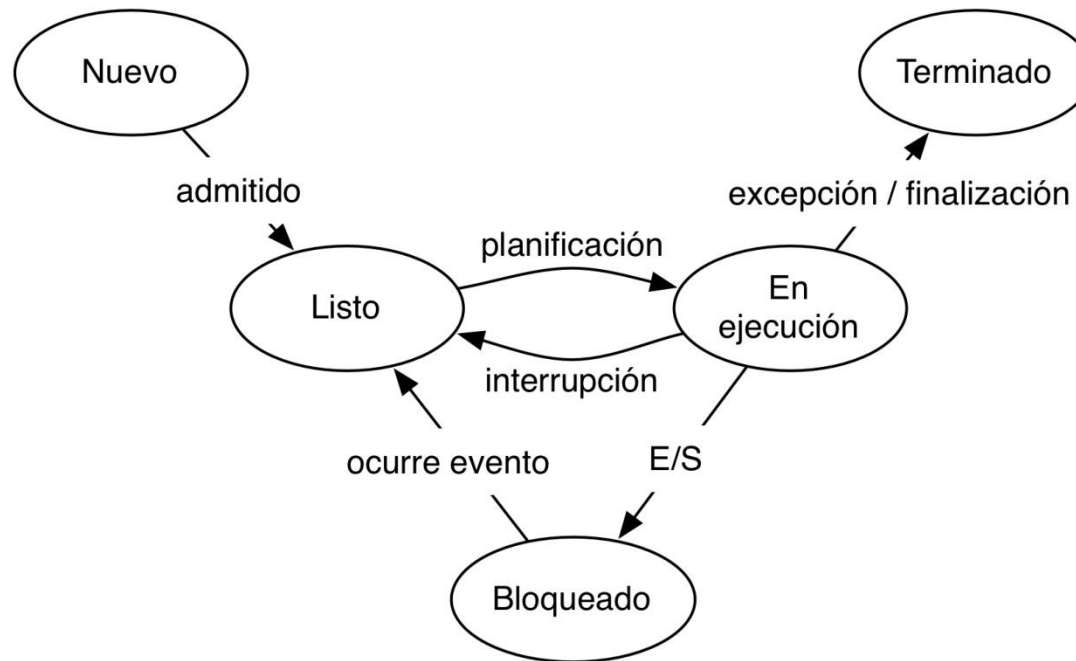
- Característica del hardware que permite al sistema operativo protegerse.
- El procesador tiene dos modos de funcionamiento
  - 0. Modo *kernel* , llamado “modo supervisor” o “modo privilegiado”.
  - 1. Modo usuario. Utilizado para la ejecución de programas de usuario.



# Estados de un proceso

- ▶ Los procesos pueden cambiar de estado a lo largo de su ejecución.
- ▶ Se definen los siguientes estados:
  - Nuevo: el proceso está siendo creado a partir del fichero ejecutable.
  - Listo: el proceso no se encuentra en ejecución aunque está preparado para hacerlo. El sistema operativo no le ha asignado todavía un procesador para ejecutarse.
  - En ejecución: el proceso se está ejecutando. El sistema operativo utiliza el mecanismo de interrupciones para controlar su ejecución. .
  - Bloqueado: el proceso está bloqueado esperando que ocurra algún suceso. Cuando ocurre el evento que lo desbloquea, el proceso no pasa directamente a ejecución sino que tiene que ser planificado de nuevo por el sistema.
  - Terminado: el proceso ha finalizado su ejecución y libera su imagen de memoria.

# Estados de un proceso



# Colas de procesos

- ▶ Los procesos se van intercambiando el uso del procesador para su ejecución de forma concurrente.
- ▶ Para ello, el sistema operativo organiza los procesos en varias colas, migrando los procesos de unas a otras:
  - **Cola de procesos:** contiene todos los procesos del sistema.
  - **Cola de procesos preparados:** contiene todos los procesos listos esperando para ejecutarse.
  - **Cola de dispositivo:** por cada dispositivo una cola diferente que contiene los procesos que están a la espera de alguna operación de E/S.

# Planificación de procesos

- ▶ El planificador es el encargado de seleccionar los movimientos de procesos entre las diferentes colas.
- ▶ Dos tipos de planificación:
  - A corto plazo:
    - Selecciona qué proceso de la cola de procesos preparados pasará a ejecución.
    - Se invoca del orden de milisegundos, cuando se produce un cambio de estado del proceso en ejecución.
    - Decisión rápida: algoritmos de planificación sencillos.
  - A largo plazo:
    - Selecciona qué procesos nuevos deben pasar a la cola de procesos preparados.
    - Se invoca con poca frecuencia, por lo que puede tomarse más tiempo en tomar la decisión.
    - Controla el grado de multiprogramación (número de procesos en memoria).

# Planificación de procesos

- ▶ Planificación a corto plazo:
  - Planificación sin desalojo o cooperativa.
    - Únicamente se cambia el proceso en ejecución si dicho proceso se bloquea o termina.
  - Planificación apropiativa.
    - Se cambia el proceso en ejecución si en cualquier momento en que un proceso se está ejecutando, otro proceso con mayor prioridad se puede ejecutar.
    - La aparición de un proceso más prioritario se puede deber tanto al desbloqueo del mismo como a la creación de un nuevo proceso.
  - Tiempo compartido:
    - cada cierto tiempo (llamado *cuanto*) se desaloja el proceso que estaba en ejecución y se selecciona otro proceso para ejecutarse.
    - Todas las prioridades de los procesos se consideran iguales.

# Cambio de contexto

- ▶ Cuando el procesador pasa a ejecutar otro proceso, el sistema operativo guarda el contexto del proceso actual y restaura el contexto del proceso que el planificador a corto plazo ha elegido ejecutar.
- ▶ Se conoce como **contexto** a:
  - Estado del proceso.
  - Estado del procesador.
  - Información de gestión de memoria.
- ▶ La salvaguarda de la información del proceso en ejecución se produce cuando hay una interrupción.
- ▶ El cambio de contexto es tiempo perdido, ya que el procesador no hace trabajo útil durante ese tiempo y su duración depende de la arquitectura en concreto del procesador.



# Gestión de procesos

- ▶ El sistema operativo es el encargado de crear los nuevos procesos siguiendo las directrices del usuario.
  - 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.
  - Cualquier proceso en ejecución (proceso hijo) siempre depende del proceso que lo creó (proceso padre), estableciéndose un vínculo entre ambos. A su vez, el nuevo proceso puede crear nuevos procesos, formándose lo que se denomina un **árbol de procesos**.

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



# Gestión de procesos

## ► Operaciones básicas:

- Creación de procesos. Cuando se crea un nuevo proceso hijo, ambos procesos, padre e hijo se ejecutan concurrentemente.
  - Ambos procesos comparten la CPU y se irán intercambiando siguiendo la política de planificación del
  - Si el proceso padre necesita esperar hasta que el hijo termine su ejecución, puede hacerlo mediante la operación *wait*.
  - Los procesos son independientes y tienen su propio espacio de memoria asignado, llamado **imagen de memoria**.
- Terminación de procesos. Al terminar la ejecución de un proceso, es necesario avisar al sistema operativo de su terminación para liberar los recursos que tenga asignados.
  - En general, es el propio proceso el que le indica al sistema mediante la operación *exit* que quiere terminar, pudiendo aprovechar para mandar información de su finalización al padre.

# Creación de procesos

- ▶ Creación de hilos en Java:
  - La clase que representa un proceso en Java es la clase *Process*.
  - Los métodos de *ProcessBuilder.start()* y *Runtime.exec()* crean un proceso nativo en el sistema operativo subyacente y devuelven una de la clase *Process* que puede ser utilizado para controlar dicho proceso.
- ▶ El método *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()*
- ▶ El método *exec(cmdarray, envp, dir)* ejecuta el comando especificado y argumentos en *cmdarray* en un proceso hijo independiente con el entorno *envp* y el directorio de trabajo especificado en *dir*

# Creación de procesos

```
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 ejecución 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("El proceso hijo finalizó
de forma incorrecta");
            System.exit(-1);
        }
    }
}
```

# Terminación de procesos

- ▶ El proceso hijo realizará su ejecución completa terminando y liberando sus recursos al finalizar.
- ▶ Esto se produce cuando el hijo realiza la operación *exit* para finalizar su ejecución.
- ▶ Un proceso padre puede además terminar de forma abrupta un proceso hijo que creó mediante la operación *destroy*
  - Esta operación elimina el proceso hijo indicado liberando sus recursos en el sistema operativo subyacente.
  - En Java, los recursos correspondientes los eliminará el *garbage collector* cuando considere.

# Terminación de procesos

```
import java.io.IOException;

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();
        } catch (IOException ex){
            System.err.println("Excepción de E/S!!");
            System.exit(-1);
        }
    }
}
```

# Comunicación de procesos

- ▶ Un proceso recibe información, la transforma y produce resultados mediante su:
  - **Entrada estándar** (*stdin*): lugar de donde el proceso lee los datos de entrada que requiere para su ejecución.
    - Normalmente es el teclado.
    - No se refiere a los parámetros de ejecución del programa.
  - **Salida estándar** (*stdout*): sitio donde el proceso escribe los resultados que obtiene.
    - Normalmente es la pantalla.
  - **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, cómo un fichero.

# Comunicación de procesos

- ▶ En Java, el proceso hijo no tiene su propia interfaz de comunicación, por lo que el usuario no puede comunicarse con él directamente.
- ▶ *Stdin*, *stdout* y *stderr* están redirigidas al proceso padre a través de los flujos de datos:
  - *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.
    - Por defecto, está conectado al mismo sitio que *stdout*.

# Comunicación de procesos

- ▶ Además de la posibilidad de comunicarse mediante flujos de datos, existen otras alternativas para la comunicación de procesos:
  - Usando *sockets*.
  - Utilizando JNI (*Java Native Interface*) para acceder desde Java a aplicaciones en otros lenguajes de programación de más bajo nivel, que pueden sacar partido al sistema operativo subyacente.
  - Librerías de comunicación no estándares entre procesos. Permiten aumentar las capacidades del estándar Java para comunicar procesos mediante:
    - Memoria compartida: se establece una región de memoria compartida entre varios procesos a la cual pueden acceder todos ellos.
    - *Pipes*: permite a un proceso hijo comunicarse con su padre a través de un canal de comunicación sencillo.
    - Semáforos: mecanismo de bloqueo de un proceso hasta que ocurra un evento.



# EJEMPLO :Comunicación de procesos utilizando un *buffer*

```
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("CMD", "/C", "DIR" ).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);
        }
    }
}
```

# Comunicación de procesos

Un proceso puede esperar recibir por su entrada estándar los datos con los que operar en un formato específico. Por ejemplo, si el proceso se crea a partir de un ejecutable en Unix, la comunicación de datos con el mismo debería producirse en UTF-8. Si los datos de entrada no contienen caracteres extraños (saltos de línea, tildes, ñ, etc.), esto no suele ser necesario, pero aun así veremos cómo puede hacerse.

# Ejemplo de comunicación con UNIX

//Importamos todos los paquetes necesarios

**import** java.io.BufferedReader;

**import** java.io.IOException;

**import** java.io.InputStream;

**import** java.io.InputStreamReader;

**import** java.util.Arrays;

**public class** UnixInteractor {

**public static void** main(String[] command){

String line;

//Preparamos el commando a ejecutar

ProcessBuilder pb = **new** ProcessBuilder(command);

pb.redirectErrorStream(**true**);

```

try {
    //Se crea el Nuevo proceso hijo
    Process shell = pb.start();
    //Se obtiene stdout del proceso hijo
    InputStream is = shell.getInputStream();
    //Se convierte el formato de UTF-8 al de un String de Java
    BufferedReader br = new BufferedReader (new InputStreamReader(is,"UTF-8"));
    System.out.println("La salida del proceso hijo" + Arrays.toString(command) + ":" );
    //Se muestra la salida del hijo por pantalla
    while ((line = br.readLine()) != null) {
        System.out.println(line);
    }
    //Cuando finaliza se cierra el descriptor
    //de salida del hijo
    is.close();
} catch (IOException e) {
    System.out.println("Error ocurrió ejecutando el comando. Descripción:" + e.getMessage());
} // catch
} //main
} //class

```

# Sincronización de procesos

- ▶ 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 resulto 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.

# Sincronización de procesos

```
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());
        }
    }
}
```

# Programación de aplicaciones multiproceso

- ▶ La programación concurrente es una forma eficaz de procesar la información al permitir que diferentes sucesos o procesos se vayan alternando en la CPU para proporcionar multiprogramación.
- ▶ El sistema operativo se encarga de proporcionar multiprogramación entre todos los procesos del sistema
  - Oculta esta complejidad tanto a los usuarios como a los desarrolladores.
- ▶ Si se pretende realizar procesos que cooperen entre sí, debe ser el propio desarrollador quien lo implemente utilizando comunicación y sincronización de procesos.

# Programación de aplicaciones multiproceso

## ► Fases:

- Descomposición funcional. Identificar previamente las diferentes cosas que debe realizar la aplicación y las relaciones existentes entre ellas.
- Partición. Distribución de las diferentes funciones en procesos estableciendo el esquema de comunicación entre los mismos.
  - La comunicación entre procesos requiere una pérdida de tiempo tanto de comunicación como de sincronización.
  - El objetivo debe ser maximizar la independencia entre los procesos minimizando la comunicación entre los mismos.
- Implementación. En este caso, Java permite únicamente métodos sencillos de comunicación y sincronización de procesos para realizar la cooperación.



# Ejercicios

**1.** Escribe una clase llamada *Ejecuta* que reciba como argumentos el comando y las opciones del comando que se quiere ejecutar. El programa debe crear un proceso hijo que ejecute el comando con las opciones correspondientes mostrando un mensaje de error en el caso de que no se realizase correctamente la ejecución. El padre debe esperar a que el hijo termine de informar si se produjo alguna anomalía en la ejecución del hijo.

# Ejercicios

2. Realiza un programa que ejecute el comando “DIR” en la consola de Windows “CMD” que capture la salida del comando DIR i la presente en la consola de depuración de nuestro proceso padre.

3. Crea un programa que ejecute el programa del ejercicio anterior (2), en este caso el programa debe ejecutar-se desde consola CMD mediante el comando “java -jar ejercicio2.jar

- ▶ Para ello deberás crear un objeto **File** que referencie al directorio donde se encuentre el fichero .jar.
- ▶ Después para establecer el directorio de trabajo para el proceso que se va a ejecutar, se debe usar el método **directory()**.
- ▶ A continuación se ejecutará el proceso y por último será necesario recoger el resultado de salida usando el método **getInputStream()** del proceso

# Ejercicios

4. Crea un programa Java llamado *LeerNombre.java* que reciba desde los argumentos de *main()* un nombre y lo visualice por pantalla. Utiliza *System.exit(1)* para una finalización correcta del programa y *System.exit(-1)* para el caso que no se hayan introducido los argumentos correctos en *main()*.

- ▶ A continuación, haz un programa parecido a del ejercicio 3 para ejecutar *LeerNombre.java*. Utiliza el método **waitFor()** para comprobar el valor de la salida del proceso que se ejecutaba. Prueba la ejecución del programa dando valor a los argumentos de *main()* y sin darle valor. ¿Qué valor devuelve **waitFor()** en un caso y en otro?

# Ejercicios

5. Crea en java un programa que visualice 5 veces la cadena que se le envía desde los argumentos de *main()*. Si no se le envía ninguna cadena que muestre un mensaje indicándolo y que finalice el programa con *System.exit(1)*.

- ▶ A continuación crea un segundo programa Java que introduzca por teclado una cadena y ejecute el programa anterior para visualizar 5 veces esa cadena.

# Ejercicios

**6.** Partiendo del ejercicio anterior, realiza los cambios necesarios para que la cadena introducida por teclado se almacene en un fichero de texto, no mostrándolo en pantalla.

Guarda los programas en otra carpeta y compílalos y ejecútalos desde la línea de comandos del DOS

**7.** Crea un programa Java que lea cadenas desde la entrada estándar hasta escribir un \*, A continuación crea otro programa que ejecute el anterior.

**8.** Realiza un programa Java que lea una cadena desde la entrada estándar y visualice en pantalla si la cadena es o no un palíndromo o si la cadena está vacía (la longitud es 0).

- ▶ Realiza un segundo programa que ejecute el anterior, debe leer la cadena desde teclado y mostrar la salida por pantalla. Transforma este ejercicio para que la cadena se obtenga de un fichero de texto, y se envíe la salida de error a un fichero

# Ejercicios

9. Escribe un programa *Aleatorios* que haga lo siguiente:

- ▶ Cree un proceso hijo que está encargado de generar números aleatorios. Para su creación puede utilizarse cualquier lenguaje de programación generando el ejecutable correspondiente. Este proceso hijo escribirá en su salida estándar un número aleatorio del 0 al 10 cada vez que reciba una petición de ejecución por parte del padre. *Nota:* no es necesario utilizar JNI, solamente crear un ejecutable y llamar correctamente al mismo desde Java.
- ▶ El proceso padre lee líneas de la entrada estándar y por cada línea que lea solicitará al hijo que le envíe un número aleatorio, lo leerá y lo imprimirá en pantalla.
- ▶ Cuando el proceso padre reciba la palabra “fin”, finalizará la ejecución del hijo y procederá a finalizar su ejecución.
- ▶ Ejemplo de ejecución:

```
ab (enter)
7
abcdef (enter)
1
Pepe (enter)
6
fin (enter)
```

# Ejercicios

**10.** Escribe una clase llamada *Mayusculas* que haga lo siguiente:

- ▶ Cree un proceso hijo.
- ▶ El proceso padre y el proceso hijo se comunicarán de forma bidireccional utilizando *streams*.
- ▶ El proceso padre leerá líneas de su entrada estándar y las enviará a la entrada estándar del hijo (utilizando el *OutputStream* del hijo).
- ▶ El proceso hijo leerá el texto por su entrada estándar, lo transformará todo a letras mayúsculas y lo imprimirá por su salida estándar. Para realizar el programa hijo se puede utilizar cualquier lenguaje de programación generando un ejecutable.
- ▶ El padre imprimirá en pantalla lo que recibe del hijo a través del *InputStream* del mismo.
- ▶ Ejemplo de ejecución:

```
hola (enter)
HOLA
mundo (enter)
MUNDO
```