

# UD 1. PROGRAMACIÓN MULTIPROCESO

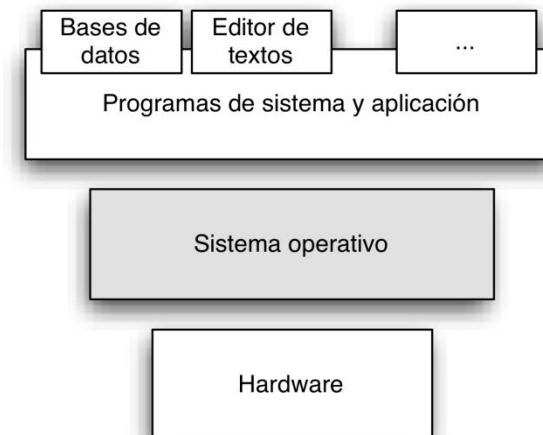
**PROGRAMACIÓN DE SERVICIOS Y PROCESOS**

Curso 2024/2025

# 1. Introducción

## Programa, Ejecutable, Proceso, Servicio o Demonio

- **Programa:** se puede considerar un programa a toda la información (tanto código como datos) almacenada en disco de una aplicación que resuelve una necesidad concreta para los usuarios.
- **Ejecutable:** fichero capaz de crear un proceso a partir de los datos de un programa.
- **Proceso:** de manera simplificada, un proceso es un programa en ejecución (aunque es importante destacar que 1 programa puede desencadenar varios procesos). Este programa en ejecución incluye (además del código y datos):
  - El contador del programa.
  - La imagen de memoria.
  - El estado del procesador.
- **Servicio (Windows) / Demonio (Linux):**
  - Proceso no interactivo que está ejecutándose continuamente en segundo plano, sin ser controlado por el usuario.
  - Suele proporcionar un servicio básico para el resto de procesos.
- **Sistema operativo:** programa encargado de:
  - Crear los procesos a partir de los ejecutables.
  - Hacer de interfaz entre el usuario y el hardware.
  - Permitir la utilización eficiente de los recursos.

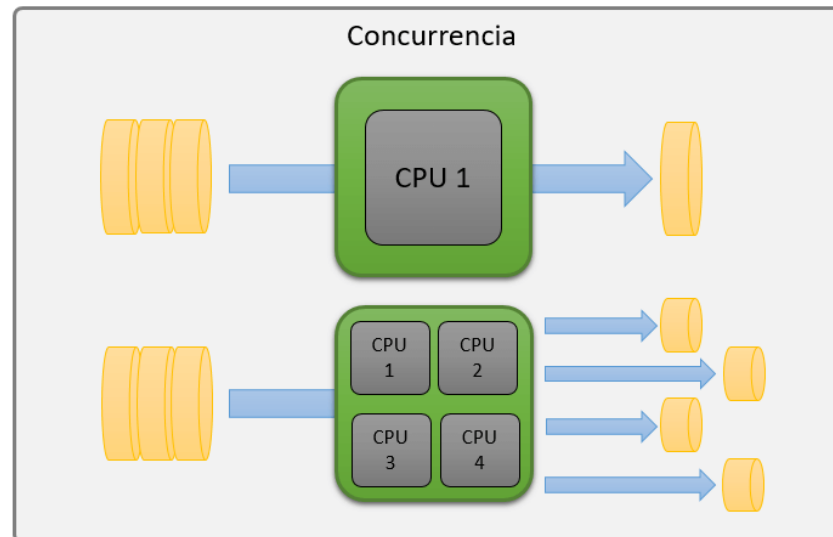


## 2. Programación concurrente, paralela y distribuida

Actualmente se pueden tener en ejecución al mismo tiempo múltiples tareas interactivas en :

### 1. 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.

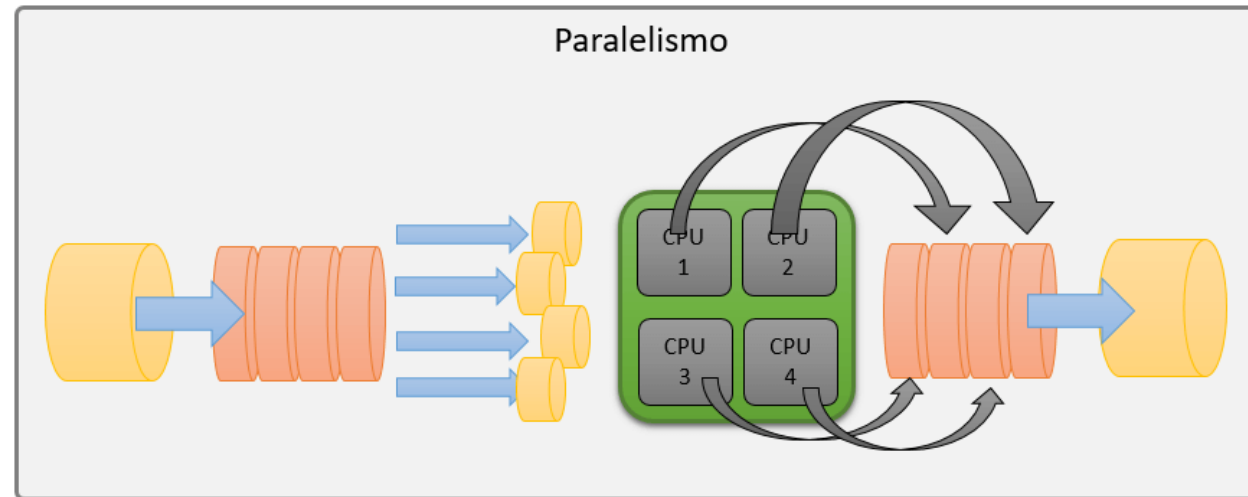


## 2. Programación concurrente, paralela y distribuida

Actualmente se pueden tener en ejecución al mismo tiempo múltiples tareas interactivas en :

### 2. 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.

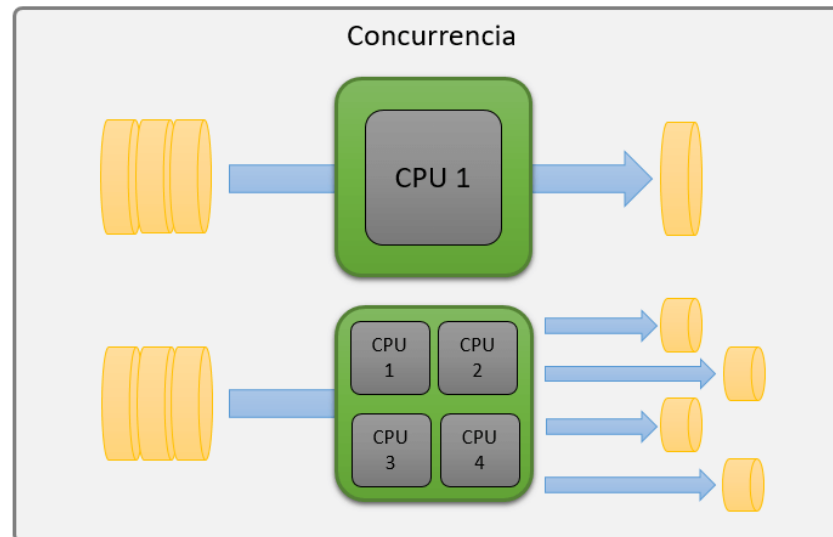


## 2. Programación concurrente, paralela y distribuida

Actualmente se pueden tener en ejecución al mismo tiempo múltiples tareas interactivas en :

### 3. Varios ordenadores distribuidos en red:

- Cada uno de los ordenadores tendrá sus propios procesadores y su propia memoria (**programación distribuida**). Por tanto, el procesamiento distribuido es aquel en el que un proceso se ejecuta en unidades de computación independientes conectadas y sincronizadas.
- 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.



## 2. Programación concurrente, paralela y distribuida

Resumen de procesamiento concurrente y paralelo:

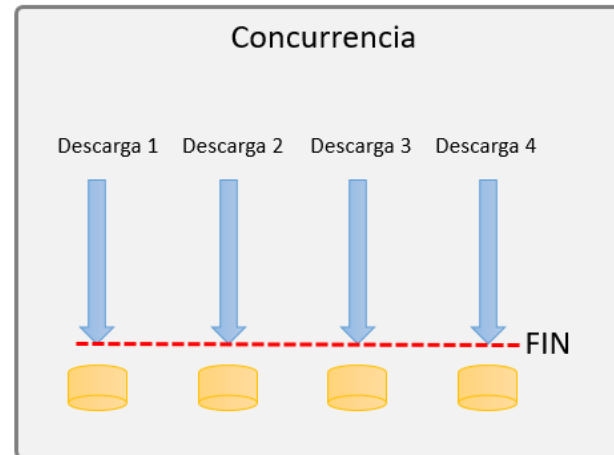
- **Procesamiento concurrente:** aquel en el que varios procesos se ejecutan en una misma unidad de proceso de manera alterna, provocando el avance simultáneo de los mismos y evitando la secuencialidad (un proceso detrás de otro).
- **Procesamiento paralelo:** aquel en el que divisiones de un proceso se ejecutan de manera simultánea en los diversos núcleos de ejecución de un procesador o en varios procesadores.

Se puede concluir, para finalizar, que el procesamiento concurrente es responsabilidad del sistema operativo mientras que el procesamiento paralelo es una responsabilidad compartida entre el sistema operativo y el programa.

## 2. Programación concurrente, paralela y distribuida

### Ejemplos de escenarios reales

Ejemplo de concurrencia: imagina una aplicación de descarga de música, en la cual puedes descargar un número determinado de canciones al mismo tiempo, cada canción es independiente de la otra, por lo que la velocidad y el tiempo que tarde en descargarse cada una no afectará al resto de canciones. Esto lo podemos ver como un proceso concurrente, ya que cada descarga es un proceso totalmente independiente del resto.

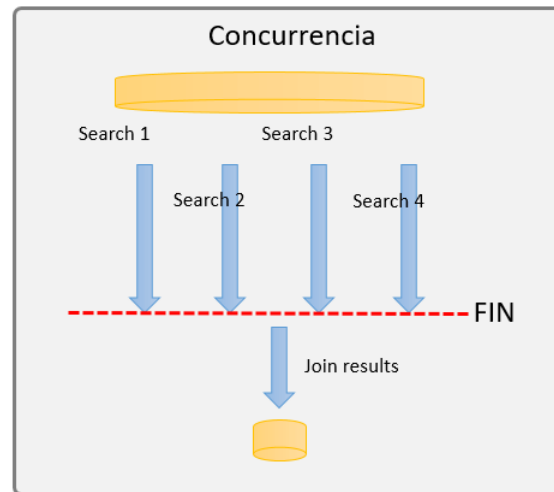


Cada descarga se procesa de forma separada e independiente. A una descarga no le importa el estado de las demás.

## 2. Programación concurrente, paralela y distribuida

### Ejemplos de escenarios reales

Ejemplo de paralelismo: imagina la clásica página de viajes, donde nos ayudan a buscar el vuelo más barato o las mejores promociones, por lo que la página debe buscar al momento en cada aerolínea el vuelo más barato, con menos conexiones, etc. Para esto puedo hacerlo de dos formas, buscar secuencialmente en cada aerolínea las mejores promociones (muy lento) o utilizar el paralelismo para buscar al mismo tiempo las mejores promociones en todas las aerolíneas.



El proceso parte de una entrada inicial (inputs) los cuales definen las características del vuelo a buscar, luego se utiliza la concurrencia para buscar en las cuatro aerolíneas al mismo tiempo. Veamos que en este proceso es indispensable que las 4 búsquedas terminen para poder arrojar un resultado. Podemos ver claramente la relación entre los 4 procesos, ya que el resultado de uno puede afectar al proceso final.

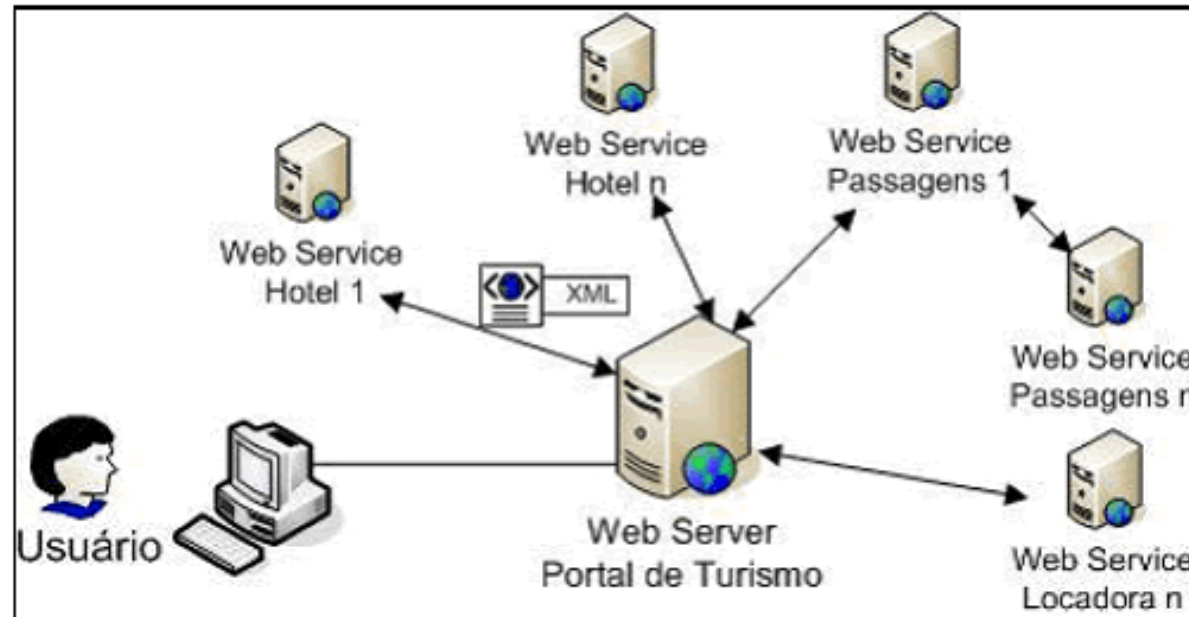
Observemos también que una vez que los cuatro procesos terminan, hay un subprocesso adicional encargado de unir los resultados y mostrar un resultado final.



## 2. Programación concurrente, paralela y distribuida

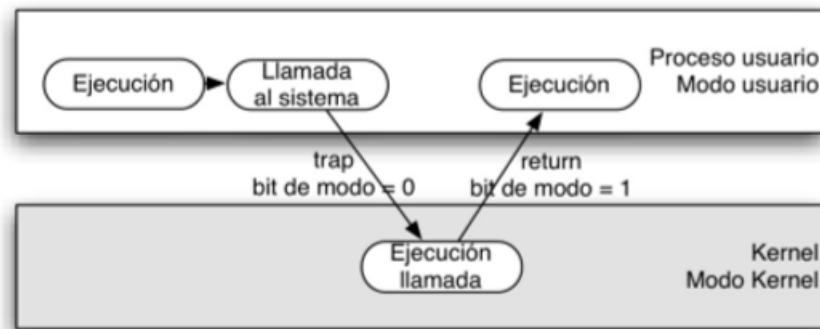
### Ejemplos de escenarios reales

Ejemplo de programación distribuida: tener una aplicación distribuida en diferentes procesos y máquinas.



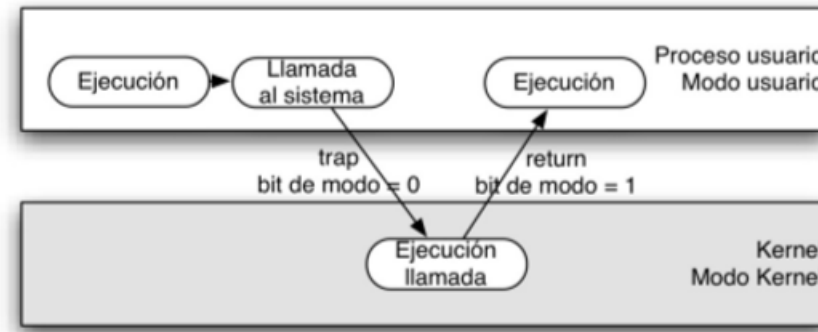
### 3. Funcionamiento básico del Sistema Operativo

- Ejecutar programas del usuario
- Hacer que el compilador sea cómodo de usar
- Utilizar los recursos del computador de forma eficiente
- Proporciona funcionalidad al resto de procesos:
  - Responde a interrupciones. Una **interrupción** es una suspensión temporal de la ejecución de un proceso, para pasar a ejecutar una rutina que trate dicha interrupción.
  - Las rutinas de tratamiento de interrupción pueden ser vistas como el código propiamente dicho del kernel. El **kernel** es el responsable de gestionar los recursos del ordenador. Funciona a través de interrupciones (trap).
    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.



### 3. Funcionamiento básico del Sistema Operativo

- Las **llamadas al sistema** son la interfaz que proporciona el kernel para que los programas de usuario puedan hacer uso de forma segura de determinadas partes del sistema.
- **Modo dual**: es una característica del hardware que permite al sistema operativo protegerse. El procesador tiene dos modos de funcionamiento indicados mediante un bit
  - Modo usuario (1): Utilizado para la ejecución de programas de usuario
  - Modo kernel (0). También llamado *modo supervisor* o *modo privilegiado*. Permite la ejecución de las instrucciones más delicadas del procesador.



## 4. Procesos

Un proceso es un programa en ejecución. Algunos ejemplos de proceso podrían ser las instancias de un navegador web, de un procesador de textos, de un entorno de desarrollo o de una máquina virtual de Java. Cada proceso está compuesto por:

- El código ejecutable.
- Los datos.
- Un contador del programa: algo que indique por dónde se está ejecutando.
- Una imagen de memoria: es el espacio de memoria que el proceso está utilizando.
- Estado del procesador: se define como el valor de los registros del procesador sobre los cuales se está ejecutando.

Los procesos están constantemente entrando y saliendo del procesador. Se denomina **contexto** a toda la información que determina el estado de un proceso en un instante dado. Es una especie de fotografía que permite quitar el proceso del procesador y restaurarlo en otro momento en el mismo estado en el que se encontraba.

Sacar a un proceso del procesador para meter a otro se conoce como cambio de contexto. Un cambio de contexto implica capturar el estado de la CPU y de sus registros, de la memoria y de la propia ejecución del proceso saliente para restaurar la información equivalente del proceso entrante y poder continuar en el punto en el que este último abandonó el procesador en el cambio de contexto anterior.

Los pasos que se han de realizar para llevar a cabo un cambio de contexto se pueden resumir en los siguientes:

- Guardar el estado del proceso actual.
- Determinar el siguiente proceso que se va a ejecutar.
- Recuperar y restaurar el estado del siguiente proceso.
- Continuar con la ejecución del siguiente proceso.

Es el sistema operativo el encargado de la gestión de los procesos, quedando en la responsabilidad del programador el crear los programas que van a dar lugar a los procesos y en manos de los usuarios el ejecutarlos.

## 4. Procesos. Planificación

**Planificador de procesos:** es el elemento del sistema operativo que se encarga de repartir los recursos del sistema entre los procesos que los demandan. De hecho, es uno de sus componentes fundamentales, ya que determina la calidad del multiproceso del sistema y, como consecuencia, la eficiencia en el aprovechamiento de los recursos.

Los objetivos del planificador son los siguientes:

- Maximizar el rendimiento del sistema.
- Maximizar la equidad en el reparto de los recursos.
- Minimizar los tiempos de espera.
- Minimizar los tiempos de respuesta.

Podemos concluir, que el objetivo del planificador es conseguir que todos los procesos terminen lo antes posible aprovechando al máximo los recursos del sistema.

Existen muchos algoritmos de planificación de los procesos. Cada sistema operativo utiliza sus propias estrategias de gestión de recursos a distintos niveles y que dichas estrategias influyen de manera directa en el funcionamiento del sistema.

Para realizar la gestión de procesos, el planificador necesita conocer el estado en el que se encuentran. En un momento dado, un proceso se encuentra en un estado de un conjunto de estados posibles.

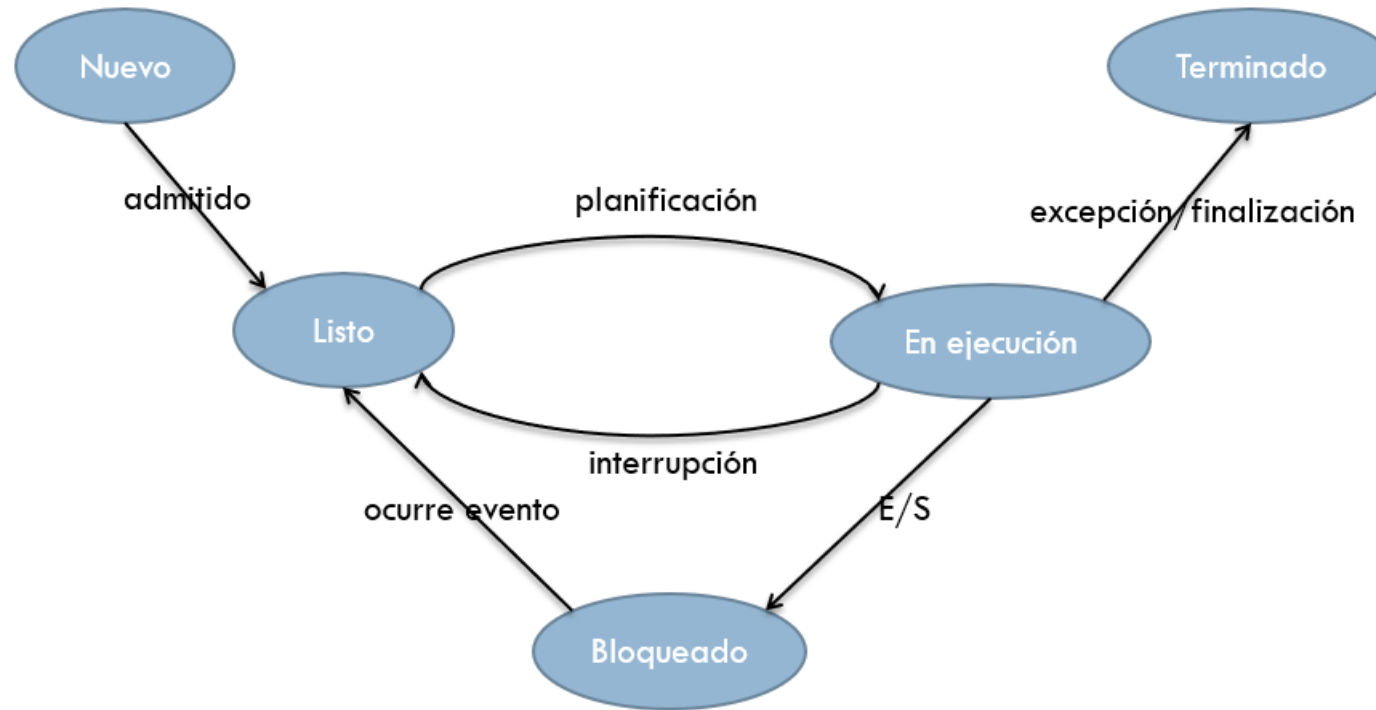
## 4. Procesos. Estados

Los estados por los que puede pasar un proceso son:

- Nuevo: técnicamente no es un estado, sino el reflejo del instante en el que se crea el proceso.
- Listo: el proceso está en memoria, preparado para ejecutarse. Está disponible, pero a la espera de que el planificador lo ponga en ejecución.
- En ejecución: el proceso se está ejecutando. Dicha ejecución es controlada por el S.O. mediante interrupciones. Si el proceso necesita un recurso se realiza una llamada al sistema. Si el proceso se ejecuta durante el tiempo máximo permitido se lanza una interrupción y si el sistema es de tiempo compartido se cambia su estado a listo.
- Bloqueado: el proceso se encuentra a la espera de que ocurra un evento externo ajeno al planificador.
- Terminado o Finalizado: al igual que el “estado” Nuevo, no es técnicamente un estado. Refleja el instante posterior a la finalización de un proceso. el

## 4. Procesos. Estados

Las posibles transiciones entre los estados se recogen en diagrama de transición de estados que representa su ciclo de vida .



## 4. Procesos. Bloque de Control de Proceso (PCB)

**Bloque de Control de Proceso (PCB):** es un registro especial donde el sistema operativo agrupa toda la información que necesita conocer respecto a un proceso particular. Cada vez que se crea un proceso el sistema operativo crea el BCP correspondiente para que sirva como descripción en tiempo de ejecución durante toda la vida del proceso.

Cada SO tiene su propio diseño de BCP. Contiene p.ej:

- Identificador único de proceso (PID) .
- Estado del proceso (p.ej: listo, en espera, bloqueado) .
- Contador de programa (PC: dirección de la próxima ejecución a ejecutar) .
- Valores de Registros de CPU .
- Espacio de direcciones de memoria .
- Información de planificación de CPU como la prioridad del proceso .
- Estadísticas del proceso: Información contable como la cantidad de tiempo de CPU y tiempo real consumido CPU y tiempo real consumido .
- Información de estado de E/S como los archivos abiertos...
- Señales pendientes de ser servidas.

Cada proceso es un elemento estanco, cada uno de ellos tiene su espacio en memoria, su tiempo de CPU asignado por el planificador y su estado de registros. No obstante, los procesos deben poder comunicarse entre sí, ya que es natural que surjan dependencias entre ellos en lo referente a las entradas y salidas de datos. Existen diversas formas de llevar a cabo la comunicación.



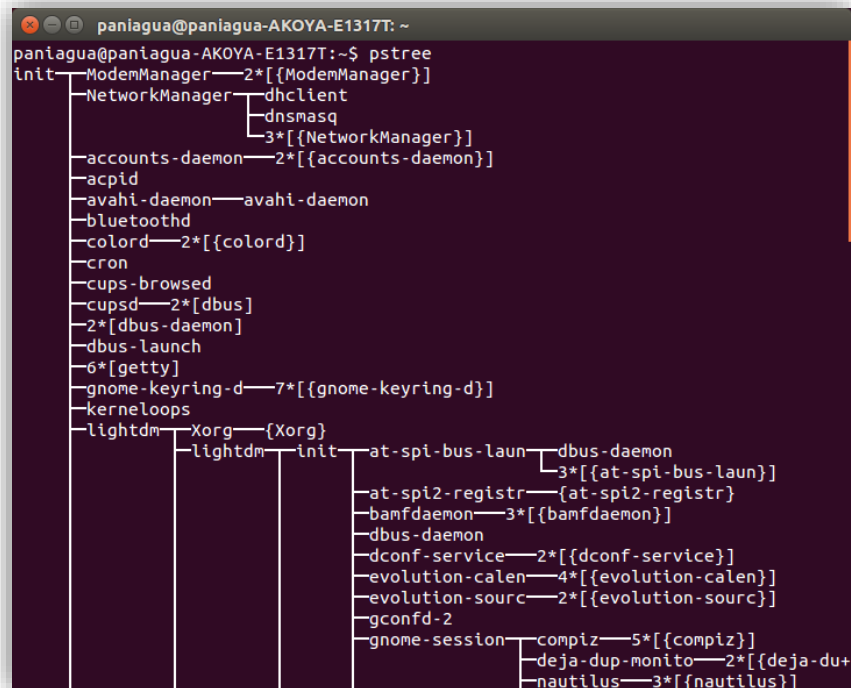
# 5. Gestión de procesos

El sistema operativo es el encargado de crear los nuevos procesos:

- 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**.

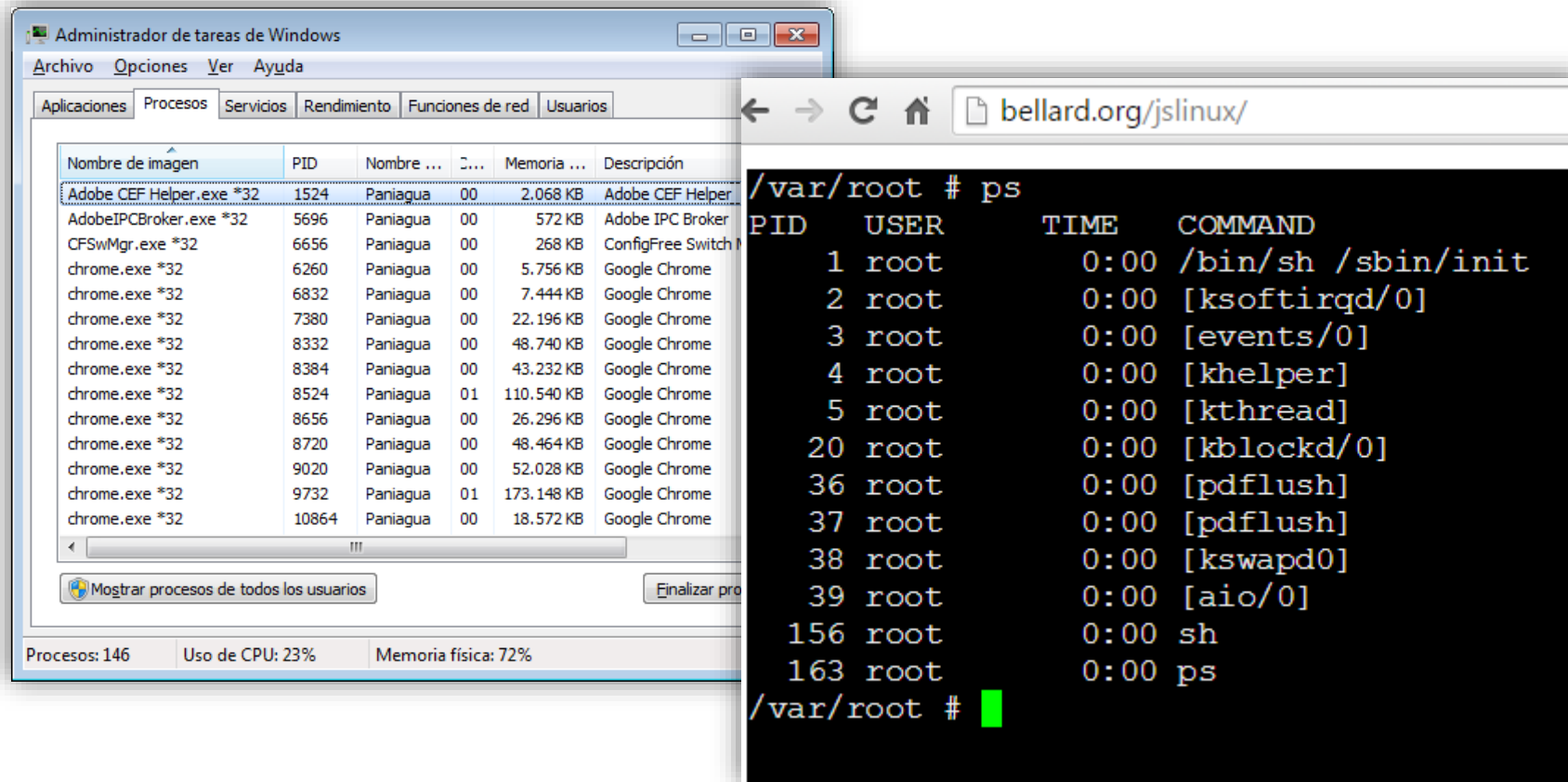
Árbol de procesos:

- Un proceso siempre es creado por otro proceso (bajo petición de un usuario o del mismo proceso).
- Un proceso en ejecución siempre depende del proceso que lo creó, formándose un árbol de procesos.



## 5. Gestión de procesos

- Los procesos se diferencian mediante un identificador (PID)



The image displays two side-by-side screenshots illustrating process management. The left screenshot shows the Windows Task Manager 'Procesos' (Processes) tab, listing various running applications with their PIDs, names, and memory usage. The right screenshot shows a Linux terminal window with the output of the 'ps' command, listing processes with their PIDs, users, times, and commands.

**Windows Task Manager - Procesos**

Nombre de imagen	PID	Nombre ...	C...	Memoria ...	Descripción
Adobe CEF Helper.exe *32	1524	Paniagua	00	2.068 KB	Adobe CEF Helper
AdobeIPCBroker.exe *32	5696	Paniagua	00	572 KB	Adobe IPC Broker
CFSwMgr.exe *32	6656	Paniagua	00	268 KB	ConfigFree Switch
chrome.exe *32	6260	Paniagua	00	5.756 KB	Google Chrome
chrome.exe *32	6832	Paniagua	00	7.444 KB	Google Chrome
chrome.exe *32	7380	Paniagua	00	22.196 KB	Google Chrome
chrome.exe *32	8332	Paniagua	00	48.740 KB	Google Chrome
chrome.exe *32	8384	Paniagua	00	43.232 KB	Google Chrome
chrome.exe *32	8524	Paniagua	01	110.540 KB	Google Chrome
chrome.exe *32	8656	Paniagua	00	26.296 KB	Google Chrome
chrome.exe *32	8720	Paniagua	00	48.464 KB	Google Chrome
chrome.exe *32	9020	Paniagua	00	52.028 KB	Google Chrome
chrome.exe *32	9732	Paniagua	01	173.148 KB	Google Chrome
chrome.exe *32	10864	Paniagua	00	18.572 KB	Google Chrome

Procesos: 146    Uso de CPU: 23%    Memoria física: 72%

**Linux Terminal - ps output**

```
/var/root # ps
PID    USER    TIME    COMMAND
1      root    0:00    /bin/sh /sbin/init
2      root    0:00    [ksoftirqd/0]
3      root    0:00    [events/0]
4      root    0:00    [khelper]
5      root    0:00    [kthread]
20     root    0:00    [kblockd/0]
36     root    0:00    [pdflush]
37     root    0:00    [pdflush]
38     root    0:00    [kswapd0]
39     root    0:00    [aio/0]
156    root    0:00    sh
163    root    0:00    ps
/var/root #
```

## 5. Gestión de procesos

- Los procesos creadores se denominan **padre**. Los creados, **hijo**. Los hijos, a su vez, pueden crear nuevos hijos. La creación de un hijo se denomina **create**.
- Padres e hijos se ejecutan de manera concurrente, compartiendo CPU según la política del planificador. Si el proceso padre debe esperar hasta que un proceso hijo termine su ejecución, deberá hacerlo mediante la operación **wait**.
- Padres e hijos utilizan espacios de memoria independientes, aunque haya diferencias en el proceso de creación de los hijos en función del sistema operativo.
  - Windows: createProcess(). Crea un nuevo proceso a partir de un programa distinto al que está en ejecución.
  - Unix: fork(). Crea un proceso hijo a partir de una copia del padre.
- Padres e hijos necesitan compartir recursos para intercambiar información: mediante **ficheros** o **memoria compartida**.
- Al terminar la ejecución de un proceso es necesario avisar al S.O. para liberar los recursos asignados. Un proceso puede notificar el fin de la ejecución mediante la operación **exit**.
- Un padre puede finalizar la ejecución de un proceso hijo, mediante la operación **destroy**.
  - La finalización de la ejecución de un proceso padre puede provocar la “terminación en cascada” de sus hijos.
- En Java:
  - Los procesos padre e hijo en la JVM no tienen por qué ejecutarse de forma concurrente.
  - No se produce terminación en cascada.

## 6. Programación de aplicaciones multiproceso en Java

En esta unidad, abordaremos la programación de aplicaciones multiproceso como la capacidad de coordinar la ejecución de un conjunto de aplicaciones para lograr un objetivo común.

Por ejemplo, si se dispone de un sistema compuesto por un conjunto de procesos que deben ejecutarse de manera individual, pero que tienen dependencias entre sí, se necesita disponer de un mecanismo de gestión y coordinación.

Las necesidades que se deben satisfacer para poder programar un sistema basado en la ejecución de múltiples procesos son las siguientes:

- Poder arrancar un proceso y hacerle llegar los parámetros de ejecución.
- Poder quedar a la espera de que el proceso termine.
- Poder recoger el código de finalización de ejecución para determinar si el proceso se ha ejecutado correctamente o no.
- Poder leer los datos generados por el proceso para su tratamiento.

En Java, la creación de un proceso se puede realizar de dos maneras diferentes:

- Utilizando la clase **java.lang.Runtime**.
- Utilizando la clase **java.lang.ProcessBuilder**.

## 6.1. Programación de aplicaciones multiproceso en Java.

### Creación de procesos con Runtime.

Toda aplicación Java tiene una única instancia de la clase **Runtime** que permite que la propia aplicación interactúe con su entorno de ejecución a través del método estático **getRuntime**. Este método proporciona un “canal” de comunicación entre la aplicación y su entorno, posibilitando la interacción con el sistema operativo a través del método **exec**.

El siguiente código Java genera un proceso en Windows indicando al entorno de ejecución (al sistema operativo) que ejecute el bloc de notas a través del programa “Notepad.exe”. En este caso, la llamada se realiza sin parámetros y sin gestionar de ninguna manera el proceso generado.

```
Runtime.getRuntime().exec(“Notepad.exe”);
```

En muchos casos, los procesos necesitan parámetros para iniciarse. El método **exec** puede recibir una cadena de caracteres y en dicha cadena, separados por espacios, se indicarán, además del programa que se desea ejecutar, los diferentes parámetros.

En el siguiente código se está ejecutando el bloc de notas indicando que “notas.txt” es el fichero que debe abrir o crear si no existe.

```
Runtime.getRuntime().exec(“Notepad.exe notas.txt”);
```

Alternativamente, se puede crear el proceso proporcionando un array de String con el nombre del programa y los parámetros.

```
String[] infoProceso={“Notepad.exe, notas.txt”};
```

```
Runtime.getRuntime().exec(infoProceso);
```

## 6.1. Programación de aplicaciones multiproceso en Java.

### Creación de procesos con Runtime.

El siguiente nivel consiste en gestionar el proceso lanzado. Para ello, se debe obtener la referencia a la instancia de la clase **Process** proporcionada por el método **exec**. Es este objeto el que proporciona los métodos para conocer el estado de la ejecución del proceso.

```
String[] infoProceso={"Notepad.exe, notas.txt"};  
  
Process proceso=Runtime.getRuntime().exec(infoProceso);
```

Si se necesita esperar a que el proceso ejecutado termine y conocer el estado en que ha finalizado dicha ejecución, se puede utilizar el método **waitFor**. Este método suspende la ejecución del programa que ha arrancado el proceso quedando a la espera de que este termine, proporcionando además el código de finalización.

```
String[] infoProceso={"Notepad.exe, notas.txt"};  
  
Process proceso=Runtime.getRuntime().exec(infoProceso);  
  
int codigoRetorno=proceso.waitFor();  
  
System.out.println("Fin de la ejecución:" +codigoRetorno);
```

## 6.1. Programación de aplicaciones multiproceso en Java.

### Creación de procesos con Runtime.

La clase `Process` representa al proceso en ejecución y permite obtener información sobre este. Los principales métodos que proporciona dicha clase son:

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

## 6.2. Programación de aplicaciones multiproceso en Java.

### Creación de procesos con ProcessBuilder.

La clase `ProcessBuilder` permite, al igual que `Runtime`, crear procesos.

La creación más sencilla de un proceso se realiza con un único parámetro en el que se indica el programa a ejecutar. Es importante saber que esta construcción no supone la ejecución del proceso.

```
new ProcessBuilder("Notepad.exe");
```

La ejecución del proceso se realiza a partir de la invocación al método **start**:

```
new ProcessBuilder("Notepad.exe").start();
```

El constructor de `ProcessBuilder` admite parámetros que serán entregados al proceso que se crea:

```
new ProcessBuilder("Notepad.exe", "datos.txt").start();
```

Al igual que ocurre con el método `exec` de la clase `Runtime`, el método `start` de `ProcessBuilder` proporciona un proceso como retorno, lo que posibilita la sincronización y gestión de este:

```
Process proceso=new ProcessBuilder("Notepad.exe", "datos.txt").start();
```

```
Int valorRetorno=proceso.waitFor();
```

```
System.out.println("Valor retorno:"+valorRetorno);
```

El método `start` permite crear múltiples subprocesos a partir de una única instancia de `ProcessBuilder`. El siguiente código crea diez instancias del bloc de notas de Windows:

```
Process pBuilder=new ProcessBuilder("Notepad.exe");
```

```
for(int i=0;i<10;i++){
```

```
    pBuilder.start();
```

```
}
```



## 6.2. Programación de aplicaciones multiproceso en Java.

### Creación de procesos con ProcessBuilder.

Además del método start, la clase ProcessBuilder dispone de métodos para consultar y gestionar algunos parámetros relativos a la ejecución del proceso.

MÉTODO	DESCRIPCIÓN
start	Inicia un nuevo proceso usando los atributos especificados.
command	Permite obtener o asignar el programa y los argumentos de la instancia de ProcessBuilder
directory	Permite obtener o asignar el directorio del trabajo del proceso.
environment	Proporciona información sobre el entorno de ejecución del proceso.
redirectError	Permite determinar el destino de la salida de errores.
redirectInput	Permite determinar el origen de la entrada estándar.
redirectOutput	Permite determinar el destino de la salida estándar.

## 6.2. Programación de aplicaciones multiproceso en Java. Creación de procesos con ProcessBuilder.

Algunos ejemplos usando los métodos anteriores son:

```
ProcessBuilder pBuilder=new ProcessBuilder("Notepad.exe", "datos.txt");  
pBuilder.directory(new File("C:/directorio_salida/"));
```

Para acceder a la información del entorno de ejecución, el método ***environment*** devuelve un objeto ***Map*** con la información proporcionada por el sistema operativo. El siguiente ejemplo muestra por pantalla el número de procesadores disponibles en el sistema:

```
ProcessBuilder pBuilder=new ProcessBuilder("Notepad.exe", "datos.txt");  
java.util.Map<String, String> env=pBuilder.environment();  
System.out.println("Número de procesadores:"+env.get("NUMBER_OF_PROCESSORS"));
```

## 6.3. Programación de aplicaciones multiproceso en Java.

### Creación de procesos. Ejemplos

EJEMPLO: Lanzar un proceso con el método `ProcessBuilder.start()`

```
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.print("Excepción de E/S!!");
            System.exit(-1);
        }catch (InterruptedException ex){
            System.err.println ("El proceso hijo finalizó de forma incorrecta");
            System.exit(-1);
        }
    }
}
```

## 6.3. Programación de aplicaciones multiproceso en Java.

### Creación de procesos. Ejemplos

EJEMPLO: Lanzar un proceso con el método `Runtime.exec()`

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

public class RunProcess2{
    public static void main(String[] args) {
        try {
            Runtime runtime = Runtime.getRuntime();
            Process process = runtime.exec("mspaint");
            //process.destroy();
        } catch (IOException ex) {
            ex.printStackTrace();
        }
    }
}
```

## 6.3. Programación de aplicaciones multiproceso en Java.

### Creación de procesos. Ejemplos

Otro ejemplo más de cómo lanzar un proceso

```
public class LanzadorProcesos {  
    public void ejecutar(String ruta){  
  
        ProcessBuilder pb;  
        try {  
            pb = new ProcessBuilder(ruta);  
            pb.start();  
        } catch (Exception e) {  
            // TODO Auto-generated catch block  
            e.printStackTrace();  
        }  
    }  
}  
/**  
 * @param args  
 */  
public static void main(String[] args) {  
    String ruta=  
        "C:\\Program Files (x86)\\Adobe\\Reader 11.0\\Reader\\AcroRd32.exe";  
    LanzadorProcesos lp=new LanzadorProcesos();  
    lp.ejecutar(ruta);  
    System.out.println("Finalizado");  
}  
}
```

## 6.3. Programación de aplicaciones multiproceso en Java.

### Creación de procesos. Ejemplos

- Un proceso puede detener a sus procesos hijos mediante *destroy*. Este proceso liberará los recursos (en Java se encarga el *garbage collector*).
- Si no se destruye el proceso, este continua hasta su finalización.

EJEMPLO: Creación de un proceso mediante Runtime para después destruirlo.

```
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);
        }
    }
}
```

## 7. Comunicación entre procesos

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

- 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, como un fichero.c .
- En la mayoría de los sistemas operativos las entradas y salidas en los procesos hijos son las mismas que en los procesos padre.
- En Java, un proceso (Process) no tiene la misma interfaz de comunicación. Las entradas y salidas se dirigen al padre a través de los siguientes *streams*:
  - OutputStream
  - InputStream
  - ErrorStream

## 7. Comunicación entre procesos

- En Java, el proceso hijo no tiene su propia interfaz de comunicación, por lo que el usuario no puede comunicarse con él directamente.
- Las entradas y salidas (stdin, stdout, stderr) se redirigen al padre a través de los siguientes *streams* o *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*.



## 7. Comunicación entre procesos

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(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);
        }
    }
}
```

## 7. Comunicación entre procesos

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

- Usando sockets.
- Utilizando JNI (Java NativeInterface) 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.