

Tema 1: Programación Multiproceso

Objetivos:

- Conocer las características de un proceso y su ejecución por el sistema operativo.
- Conocer las características y diferencias de la programación concurrente, paralela y distribuida.
- Crear procesos en Linux y utilizar clases Java para crear procesos.
- Desarrollar programas que ejecuten tareas en paralelo.

1.- Conceptos básicos

Programa: Se puede considerar un programa a toda la información (tanto código como datos) de una aplicación que resuelve una necesidad concreta para los usuarios.

Proceso: Podemos definir “Proceso” como un programa en ejecución. Este concepto no se refiere únicamente al código y a los datos, sino que incluye todo lo necesario para su ejecución. Esto incluye tres cosas:

- **Un contador de programa:** Algo que indique por dónde se está ejecutando el proceso.
- **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.

Ejecutable: *Un fichero ejecutable contiene la información necesaria para crear un proceso a partir de los datos almacenados de un programa; es decir, ejecutable es aquel fichero que permite poner el programa en ejecución como proceso.*

Sistema operativo: Programa que hace de intermediario entre el usuario y las aplicaciones que utiliza y el hardware del ordenador. Entre sus objetivos, podemos destacar:

- Ejecutar los programas del usuario.
- Hacer que el computador sea cómodo de usar.
- Utilizar los recursos del computador de forma eficiente.

Demonio: Proceso no interactivo que está ejecutándose continuamente en segundo plano, es decir, *es un proceso controlado por el sistema sin ninguna intermediación del usuario*. Suelen proporcionar un servicio básico para el resto de procesos.

En un sistema operativo **multiproceso o multitarea** se puede ejecutar más de un proceso (programa) a la vez, dando la sensación de que cada proceso es el único que se está ejecutando. La única forma de ejecutar varios procesos simultáneamente es tener varias CPUs (bien en una máquina o en varias). En los sistemas operativos con una única CPU se va alternando la ejecución de los procesos, es decir, se quita un proceso de la CPU, se ejecuta otro y se vuelve a colocar el primero sin que se entere de nada. *Es el sistema operativo quien decide parar la ejecución de un proceso, por ejemplo, porque ha consumido su tiempo de CPU, y arrancar la de otro.* **Cuando se suspende temporalmente la ejecución de un proceso debe reanudarse posteriormente en el mismo estado en que se encontraba cuando se paró, esto implica que toda la información**

referente al proceso debe almacenarse en alguna parte. A esto se le denomina **programación concurrente**, esta no mejora el tiempo de ejecución global de los programas ya que se ejecutan intercambiando unos por otros en el procesador; sin embargo, permiten que parezca que varios programas se ejecutan al mismo tiempo.

La **programación multiproceso** tiene en cuenta la posibilidad de que múltiples procesos puedan estar ejecutándose simultáneamente sobre el mismo código de programa. *Es decir desde una misma aplicación podemos realizar varias tareas de forma simultánea, o lo que es lo mismo, podemos dividir un proceso en varios subprocesos.*

1.2.- Programación concurrente

La **computación concurrente** permite la ejecución al mismo tiempo de múltiples tareas interactivas. Es decir, permite realizar varias cosas al mismo tiempo, como escuchar música, visualizar la pantalla, imprimir documentos, etc. Dichas tareas se pueden ejecutar en:

- **Un único procesador (multiprogramación).** Si solo existe un solo procesador, solamente un proceso puede estar en un momento determinado en ejecución. (**Programación concurrente**).
- **Varios núcleos en un mismo procesador (multitarea).** La existencia de varios núcleos o cores en un ordenador. Cada núcleo podría estar ejecutando una instrucción diferente al mismo tiempo. Es el sistema operativo quien se encarga de planificar los trabajos que se ejecutan en cada núcleo y cambiar unos por otros para generar multitarea. En este caso, todos los cores comparten la misma memoria por lo que es posible utilizarlos de forma coordinada mediante lo que se conoce como **programación paralela**. La programación paralela permite mejorar el rendimiento de un programa si este se ejecuta de forma paralela en diferentes núcleos ya que permite que se ejecuten varias instrucciones a la vez. Cada ejecución en cada core será una **tarea** (o “**hilo de ejecución**”) del mismo programa pudiendo cooperar entre si.
- **Varios ordenadores distribuidos en red.** Cada uno de los ordenadores tendrá sus propios procesadores y su propia memoria. La gestión de los mismos forma parte de lo que se denomina **programación distribuida**. La programación distribuida posibilita la utilización de un gran número de dispositivos (ordenadores) de forma paralela, lo que permite alcanzar grandes mejoras en el rendimiento de la ejecución de programas distribuidos. Sin embargo, como cada ordenador posee su propia memoria, imposibilita que los procesos puedan comunicarse compartiendo memoria, teniendo que utilizar otros esquemas de comunicación más complejos y costosos a través de la red que los interconectan.

1.3.- Funcionamiento de un sistema operativos

La parte del sistema operativo que realiza la funcionalidad básica se denomina **kernel**. A todo el resto del sistema operativo se le denomina **programas del sistema**. El kernel es el responsable de gestionar los recursos del ordenador, permitiendo su uso a través de llamadas al sistema.

En general el **kernel** funciona a base de interrupciones. Una **interrupción** es una suspensión temporal de la ejecución de un proceso, para pasar a ejecutar una rutina que trate dicha interrupción.

Esta rutina será dependiente del sistema operativo. Cuando finaliza la rutina, se reanuda la ejecución del proceso en el mismo lugar donde se quedó cuando fue interrumpido.

Cuando salta una interrupción se transfiere el control a la rutina de tratamiento de la interrupción. Así, las rutinas de tratamiento de interrupción pueden ser vista como el código propiamente dicho del kernel.

Las **llamadas al sistema** son **interfaz** que proporciona el kernel para que los programas de usuario puedan hacer uso de forma segura de determinadas partes del sistema. Los errores de un programa podrían afectar a otros programas o al propio sistema operativo, por lo que para asegurar su ejecución de la forma correcta, el sistema implementa una interfaz de llamadas para evitar que ciertas instrucciones peligrosas sean ejecutadas directamente por programas de usuario.

El **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) : Se suele llamar “modo supervisor” o “modo privilegiado”. En este modo se pueden ejecutar las instrucciones del procesador más delicadas.

1.4.- Procesos

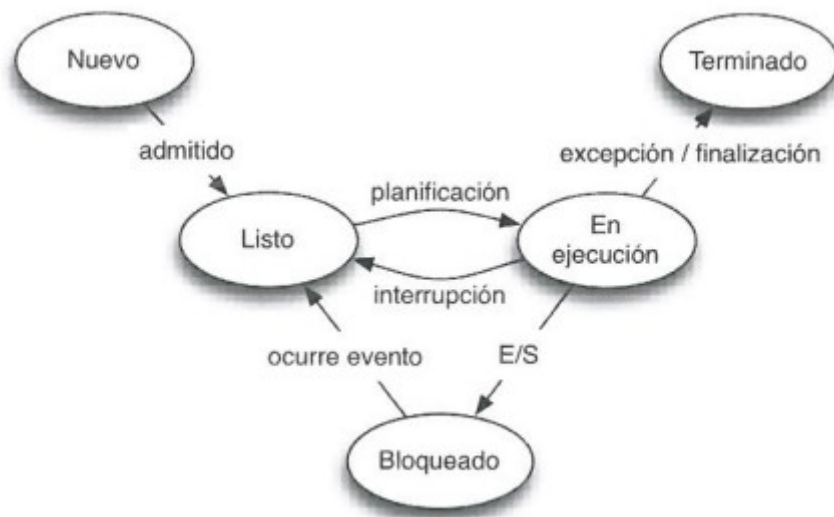
El sistema operativo es el encargado de poner en ejecución y gestionar los procesos. Para su correcto funcionamiento, a lo largo de su ciclo de vida, los procesos pueden cambiar de estado. Es decir, a medida que se ejecuta un proceso, dicho proceso pasará por varios estados. El cambio de estado también se producirá por la intervención del sistema operativo.

1.4.1.- Estado de un proceso

Los estados de un proceso son:

- **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. El planificador del sistema operativo es el responsable de seleccionar que proceso está en ejecución, por lo que es el que indica cuando el proceso pasa a ejecución.
- **En ejecución:** El proceso se está ejecutando. El sistema operativo utiliza el mecanismo de interrupciones para controlar su ejecución. Si el proceso necesita un recurso, incluyendo la realización de operaciones de entrada/salida (E/S), llamará a la llamada del sistema correspondiente. Si un proceso en ejecución se ejecuta durante el tiempo máximo permitido por la política del sistema, salta un temporizador que lanza una interrupción. En este último caso, si el sistema es de tiempo compartido, lo para y lo pasa al estado “Listo”, seleccionando otro proceso para que continúe su ejecución.
- **Bloqueado:** El proceso está bloqueado esperando que ocurra algún suceso (esperando por una operación de E/S, bloqueado para sincronizarse con otros procesos, etc.). 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. Para terminar un proceso, el mismo debe llamar al sistema para indicárselo o puede ser el propio sistema el que finalice el proceso mediante una excepción (una interrupción especial).



1.4.2.- Colas de procesos

Uno de los objetivos del sistema operativo es la **multiprogramación**, es decir, admitir varios procesos de memoria para maximizar el uso del procesador. Los procesos se irán intercambiando el uso del procesador para su ejecución de forma concurrente. Para ello, el sistema operativo organiza los procesos en varias colas, migrándolos de unas colas a otras:

- Una **cola de procesos** que contiene todos los procesos del sistema.
- Una **cola de procesos preparados** que contiene todos los procesos listo esperando para ejecutarse.
- Varias **colas de dispositivo** que contienen los procesos que están a la espera de alguna operación de E/S.

1.4.3.- Planificación de procesos

El planificador es el encargado de seleccionar los movimientos de procesos entre las diferentes colas. Existen dos tipos de planificación:

- **A corto plazo:** selecciona qué proceso de la cola de procesos preparados pasará a ejecución. Se invoca muy frecuentemente por lo que debe ser muy rápido en la decisión. Esto implica que los algoritmos sean muy sencillos.
 - **Planificación sin desalojo o cooperativo:** Únicamente se cambia el proceso en ejecución si dicho proceso se bloquea o termina.

- **Planificación apropiativa:** Además de los casos de planificación cooperativa, 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. En este caso, todas las prioridades de los procesos se consideran iguales.
- **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).

1.4.4.- Cambio de contexto.

Cuando el procesador pasa a ejecutar otro proceso, el sistema operativo debe de guardar el contexto del proceso actual y restaurar el contexto del proceso que el planificador ha elegido ejecutar.

Se conoce como **contexto** a:

- *Estado del proceso.*
- *Estado del procesador:* valores de los diferentes registros del procesador.
- *Información de gestión de memoria:* espacio de memoria reservada para el proceso.

1.5.- Gestión de procesos

1.5.1.- Árbol de procesos

Aunque el responsable de crear el nuevo proceso es el sistema operativo, ya que es quien 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.

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 ese proceso, se crea el resto de procesos de forma jerárquica, estableciendo padres, hijos, abuelos, etc.

El **BCP** es una estructura de datos llamada **Bloque de Control de Proceso** donde se almacena información acerca de un proceso.

- Identificación del proceso (PID). Cada proceso que se inicia es referenciado por un identificador único.
- Estado del proceso.
- Contador de programa.

- Registros de la CPU.
- Información de planificación de CPU como la prioridad del proceso.
- Información de gestión de memoria.
- Información contable como la cantidad de tiempo de CPU y tiempo real consumido.
- Información de estado de E/S como la lista de dispositivos asignados, archivos abiertos, etc.

Ejemplo:

Mediante el comando “ps” (process status) de linux, podemos ver parte de la información asociada a cada procesos

```

juan@juan:~$ ps
  PID TTY          TIME CMD
 5518 pts/2        00:00:00 bash
 5529 pts/2        00:00:00 ps
juan@juan:~$

```

Nos muestra dos procesos que se están ejecutando, uno es el shell y el otro es la ejecución de la orden “ps”.

- **PID:** Identificador del proceso.
- **TTY:** Terminal asociado del que lee y al que escribe. Si no hay aparece interrogación. En caso de que una TTY llame a más terminales se crean terminales virtuales llamados PTS. Para conocer en que terminal estamos, usaremos los comandos tty, ps

```

juan@juan:~$ tty
/dev/pts/2
juan@juan:~$

```

- **TIME:** tiempo de ejecución asociado , es la cantidad total de tiempo de CPU que el proceso ha utilizado desde que nació.
- **CMD:** nombre del proceso.

Otra información adicional.

```

root@juan:/# ps -f
UID          PID  PPID  C  STIME TTY          TIME CMD
root         5972   5518  0  19:09 pts/2        00:00:00 sudo su
root         5973   5972  0  19:09 pts/2        00:00:00 su
root         5974   5973  0  19:09 pts/2        00:00:00 bash
root         6023   5974  0  19:12 pts/2        00:00:00 ps -f

```

- **UID** : Nombre del usuario.
- **PPID**: PID del padre de cada proceso.
- **C**: Porcentaje de recursos de CPU utilizado por el proceso.
- **STIME**: Hora de inicio del proceso.

La orden `ps -AF` muestra todos los procesos activos con todos los detalles

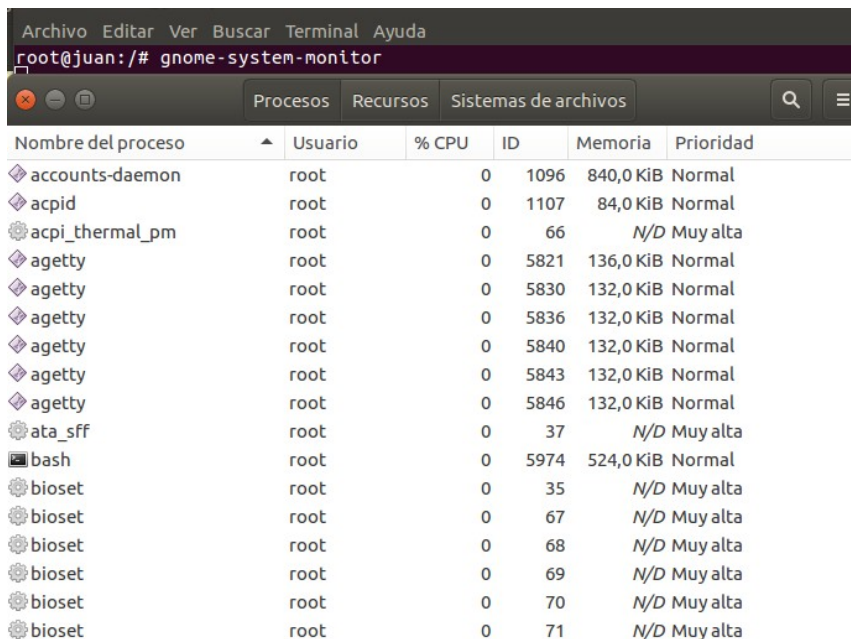
```

root@juan:/# ps -AF
UID      PID  PPID  C   SZ   RSS  PSR  STIME TTY      TIME  CMD
root      1    0    0 46378 6156   1 16:37 ?        00:00:00 /sbin/init splas
root      2    0    0    0    0    0 16:37 ?        00:00:00 [kthreadd]
root      3    2    0    0    0    0 16:37 ?        00:00:00 [ksoftirqd/0]
root      5    2    0    0    0    0 16:37 ?        00:00:00 [kworker/0:0H]
root      7    2    0    0    0    1 16:37 ?        00:00:00 [rcu_sched]
root      8    2    0    0    0    0 16:37 ?        00:00:00 [rcu_bh]
root      9    2    0    0    0    0 16:37 ?        00:00:00 [migration/0]
root     10    2    0    0    0    0 16:37 ?        00:00:00 [watchdog/0]
root     11    2    0    0    0    1 16:37 ?        00:00:00 [watchdog/1]
root     12    2    0    0    0    1 16:37 ?        00:00:00 [migration/1]
root     13    2    0    0    0    1 16:37 ?        00:00:00 [ksoftirqd/1]
root     15    2    0    0    0    1 16:37 ?        00:00:00 [kworker/1:0H]
root     16    2    0    0    0    2 16:37 ?        00:00:00 [watchdog/2]
root     17    2    0    0    0    2 16:37 ?        00:00:00 [migration/2]
root     18    2    0    0    0    2 16:37 ?        00:00:00 [ksoftirqd/2]
root     20    2    0    0    0    2 16:37 ?        00:00:00 [kworker/2:0H]
root     21    2    0    0    0    3 16:37 ?        00:00:00 [watchdog/3]
root     22    2    0    0    0    3 16:37 ?        00:00:00 [migration/3]
root     23    2    0    0    0    3 16:37 ?        00:00:00 [ksoftirqd/3]
root     25    2    0    0    0    3 16:37 ?        00:00:00 [kworker/3:0H]
root     26    2    0    0    0    2 16:37 ?        00:00:00 [kdevtmpfs]

```

- **SZ**: Tamaño virtual de la imagen del proceso.
- **RSS**: Tamaño de la parte residente en memoria en kilobytes.
- **PSR**: Procesador (núcleo) que el proceso tiene actualmente asignado.

Con el comando “ `sudo gnome-system-monitor`” podemos acceder a la interfaz gráfica que nos muestra información sobre los procesos que se están ejecutando.



Nombre del proceso	Usuario	% CPU	ID	Memoria	Prioridad
accounts-daemon	root	0	1096	840,0 KiB	Normal
acpid	root	0	1107	84,0 KiB	Normal
acpi_thermal_pm	root	0	66	N/D	Muy alta
agetty	root	0	5821	136,0 KiB	Normal
agetty	root	0	5830	132,0 KiB	Normal
agetty	root	0	5836	132,0 KiB	Normal
agetty	root	0	5840	132,0 KiB	Normal
agetty	root	0	5843	132,0 KiB	Normal
agetty	root	0	5846	132,0 KiB	Normal
ata_sff	root	0	37	N/D	Muy alta
bash	root	0	5974	524,0 KiB	Normal
bioset	root	0	35	N/D	Muy alta
bioset	root	0	67	N/D	Muy alta
bioset	root	0	68	N/D	Muy alta
bioset	root	0	69	N/D	Muy alta
bioset	root	0	70	N/D	Muy alta
bioset	root	0	71	N/D	Muy alta

1.5.2.- 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 se le llama **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 (espera)**.

Padres e hijos son procesos que aunque tengan un vínculo especial, usan espacio de memoria independientes. Estos pueden compartir recursos para intercambiarse información. *Estos recursos pueden ir desde ficheros abiertos hasta zonas de memoria compartida*. La memoria compartida es una zona 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 un proceso, es necesario avisar al sistema operativo de su terminación para que este libere los recursos asignados si es posible. Es el propio proceso el que le indica al sistema operativo mediante una operación llamada **exit** que quiere terminar y aprovechar para mandar información de su finalización al proceso padre.

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.

1.5.2.1- Creación de procesos (operación create)

El paquete de **java.lang** incluye clases para la gestión de procesos como son **Process** y **Runtime**. Cada aplicación Java dispone de la clase **Runtime** que representa el entorno de ejecución de la aplicación:

- **static Runtime getRuntime()**: que devuelve el objeto **Runtime** asociado con la aplicación Java.
- **Process exec (String comando)**: ejecuta la orden en un proceso separado. Devuelve un objeto **Process**, que se puede utilizar para controlar la interacción del programa Java con el nuevo proceso de ejecución.

Ejemplo1 : Proceso que abre el bloc de notas . Este proceso se debe de ejecutar en windows


```
package EjemploProcesos;

package EjemploProcesos;

import static java.lang.Runtime.getRuntime;

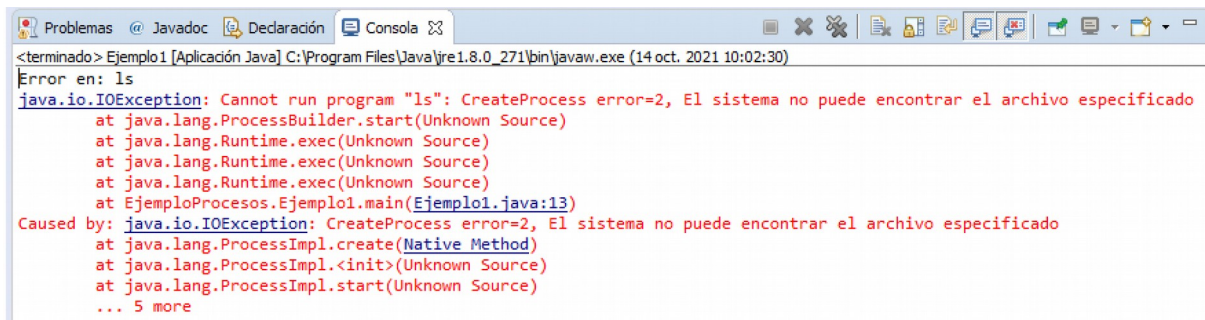
public class Ejemplo1 {
    public static void main(String[] args) {

        Runtime r = getRuntime();// Crea un objeto Runtime para ejecutar la aplicación
        // Establecemos el comando. En este caso es llamar al programa que abre la
        String comando = "notepad.exe";
        Process p; // Creamos un objeto Process para lanzar un nuevo proceso
        try {
            p = r.exec(comando); //Ejecuta la orden del nuevo proceso
        } catch (Exception e) {
            System.out.println("Error en: "+comando);
            e.printStackTrace();
        }
    }
}
```

El resultado es que se abre el bloc de notas.

Si el comando fuera “ls” el programa no mostraría nada ya que la salida del proceso invocado no se dirige a la pantalla sino al propio **Java**. Para leer la salida que devuelve **exec()** del Runtime tenemos que usar el objeto **Process**.

Si el comando fuera “ls” el programa no mostraría nada ya que la salida del proceso invocado no se dirige a la pantalla sino al propio **Java**. Para leer la salida que devuelve **exec()** del Runtime tenemos que usar el objeto **Process**.



El método **getInputStream()** nos permite leer el stream de salida del proceso.

Ejemplo 2: Lectura del flujo de salida de un proceso.

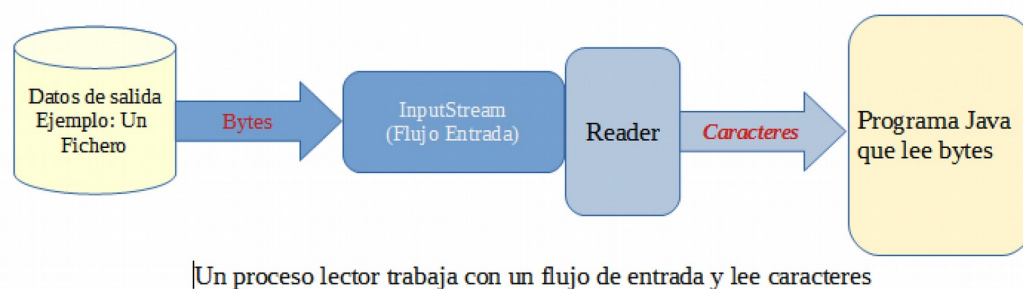
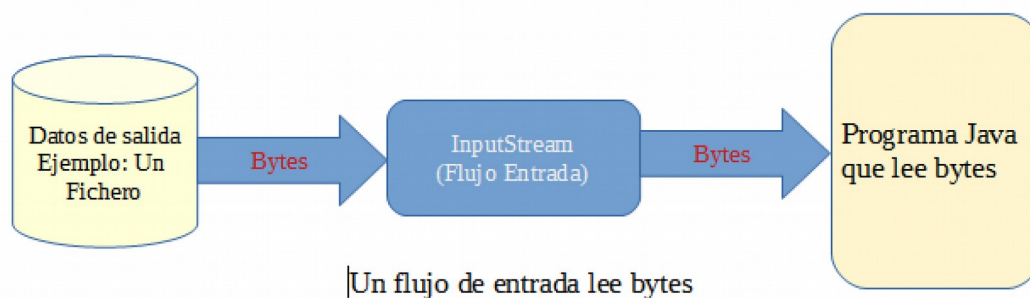
```

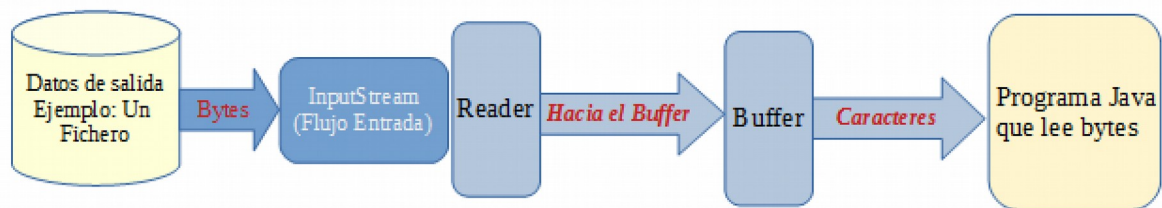
package EjemploProcesos;

import java.io.*;
public class Ejercicio2 {

    public static void main(String[] args) {
        Runtime r = Runtime.getRuntime();
        String comando = "NOTEPAD";
        Process p = null;
        try {
            p = r.exec(comando);
            // Aquí procedemos a leer lo que devuelve el comando
            InputStream is = p.getInputStream();//llamada al método
            BufferedReader br;// Creamos buffer
            br = new BufferedReader(new InputStreamReader(is)); //Le pasamos al buffer
            el flujo de salida
            String linea;
            while ((linea=br.readLine())!=null) //lee una línea del flujo
                System.out.println(linea);
            br.close();
        } catch (Exception e) {
            System.out.println("Error en: "+comando);
            e.printStackTrace();
        }
    }
}

```





Un lector con búfer almacena previamente los caracteres para mejorar el rendimiento.

Para mejorar el programa deberíamos esperar a que el proceso lanzado termine para eso utilizamos el método **waitFor()**, que devolverá 0 si el subprocesso termina correctamente.

Para mejorar el programa deberíamos esperar a que el proceso lanzado termine para eso utilizamos el método **waitFor()**, que devolverá 0 si el subprocesso termina correctamente.

Si se produce un problema: supongamos que en el ejemplo anterior se intenta ejecutar algún comando erróneo; saldrá como valor de salida 1. Si queremos tener más información sobre el error de salida debemos capturar y tratar el error:

```

package EjemploProcesos;

import static java.lang.System.out;
import java.io.IOException;
import java.io.InputStream;
import java.io.BufferedReader;
import java.io.InputStreamReader;

public class Ejercicio2 {

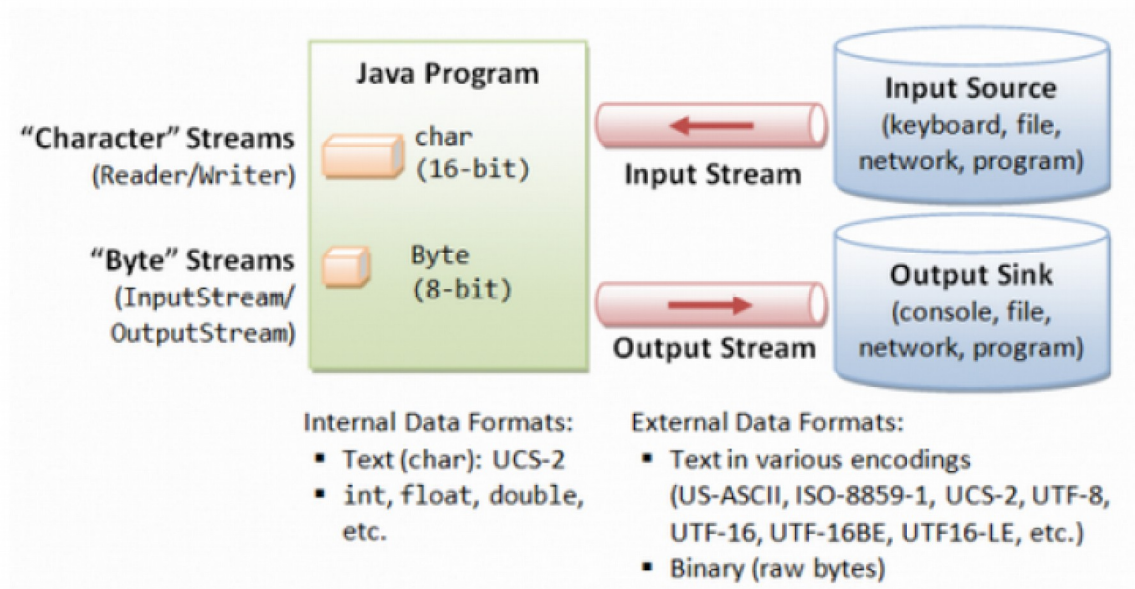
    public static void main(String[] args) {
        Runtime r = Runtime.getRuntime();
        String comando = "NOTEPAD";
        Process p = null;
        try {
            p = r.exec(comando);
            // Aquí procedemos a leer lo que devuelve el comando
            InputStream is = p.getInputStream();
            BufferedReader br;
            br = new BufferedReader(new InputStreamReader(is));
            String linea;
            while ((linea=br.readLine())!=null) //lee una línea
            {
                out.println(linea);
            }
            br.close();
        } catch (Exception e) {

```

```

        out.println("Error en: "+comando);
        e.printStackTrace();
    }
    try {
        InputStream er = p.getErrorStream();
        BufferedReader brer = new BufferedReader (new InputStreamReader(er));
        String linea2 = null;
        while((linea2 = brer.readLine()) != null) {
            linea2 = brer.readLine();
            System.out.println("ERROR >" + linea2);
        }
    } catch (IOException ioe) {
        ioe.printStackTrace();
    }
}
}
}

```



Ejemplo: Proceso que envía 5 saludos (Unsaludo.java)

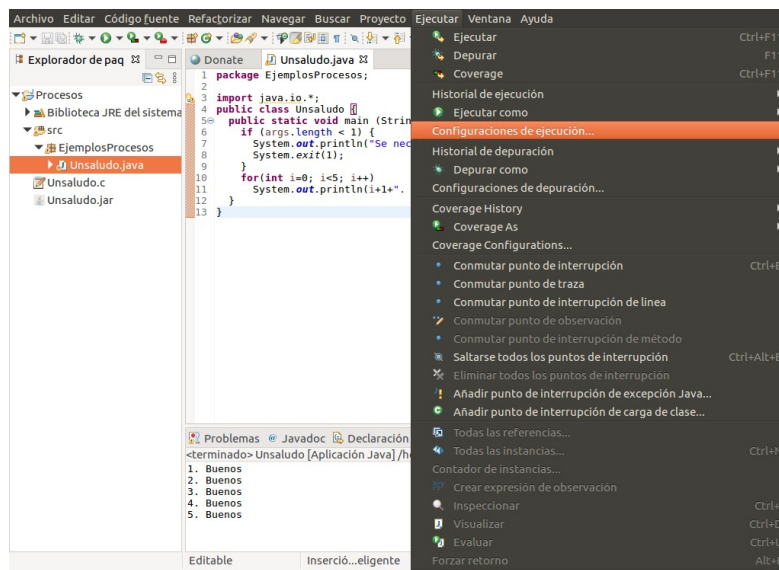
```

package EjemplosProcesos;

import java.io.*;

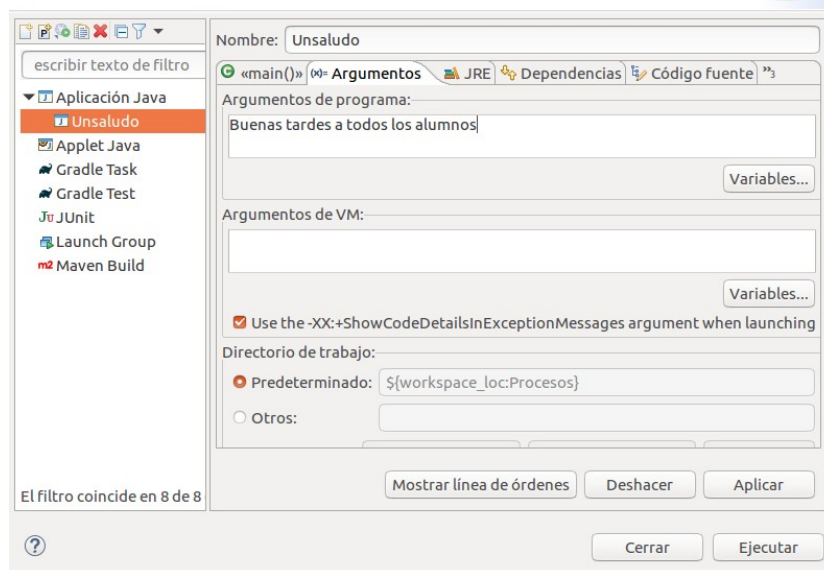
public class Unsaludo {
    public static void main (String[] args){
        if (args.length < 1) {
            System.out.println("Se necesita que me indiques un saludo.");
            System.exit(1);
        }
        for(int i=0; i<5; i++)
            System.out.println(i+1+" . "+args[0]);
    }
}

```

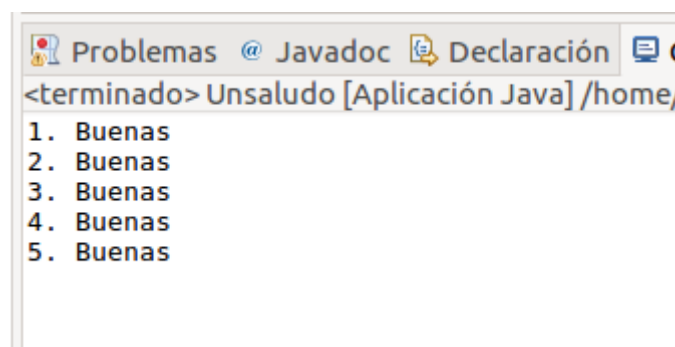


Crear, gestionar y ejecutar configuraciones

Ejecutar una aplicación Java



Ejecutamos:



NOTA: La clase `Process` tiene el método `getOutputStream()` que permite escribir en el flujo de entrada del proceso, de esta forma podemos enviar datos a un comando.

Ejemplo 3: (en el sistema windows)

El comando “date” nos devuelve la fecha actual y nos pide que escribamos de forma interactiva una nueva fecha si queremos cambiarla.

Utilizando la clase “`getOutputStream()`” de Java podemos escribir en el Stream de entrada del proceso de forma que podemos enviar datos a un comando.

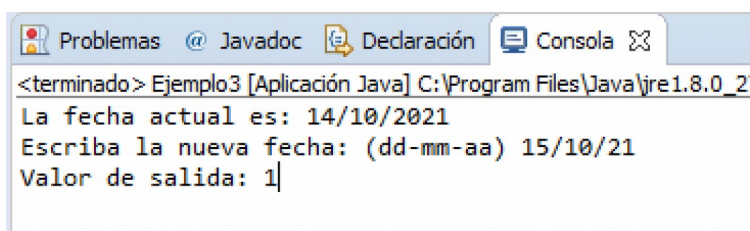
Realizar un programa que ejecute el comando `date` y le de valores de 15-10-2021

```
package EjemploProcesos;

import java.io.*;

public class Ejemplo3 {
    public static void main(String[] args) {
        Runtime r = Runtime.getRuntime();
        String comando="CMD /C DATE ";
        Process p=null;
        try {
            p=r.exec(comando);
            //escritura -- enviamos entrada a date
            OutputStream os = p.getOutputStream();
            os.write("15/10/21".getBytes());
            os.flush(); // vacía buffer de salida

            //lectura -- obtiene la salida de date
            InputStream is = p.getInputStream();
            BufferedReader br = new BufferedReader (new InputStreamReader(is));
            String linea;
            while((linea=br.readLine())!=null)
                System.out.println(linea);
            br.close();
        } catch (Exception e) { e.printStackTrace();}
        // comprobación de error 0 bien; 1 mal.
        int exitVal;
        try {
            exitVal = p.waitFor();
            System.out.println("Valor de salida: " + exitVal);
        } catch (InterruptedException e){
            e.printStackTrace();
        }
    }
}
```



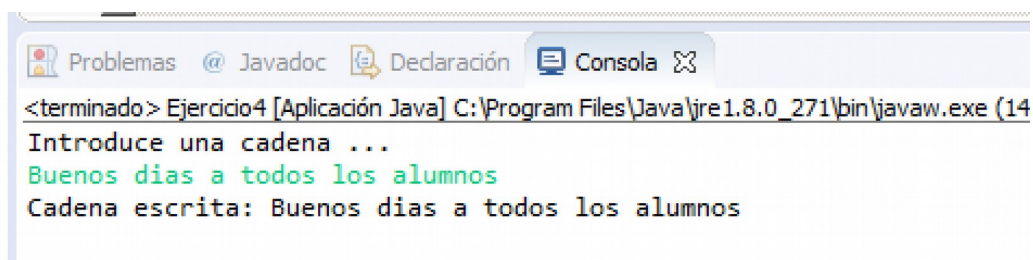
```
<terminado> Ejemplo3 [Aplicación Java] C:\Program Files\Java\jre1.8.0_21
La fecha actual es: 14/10/2021
Escriba la nueva fecha: (dd-mm-aa) 15/10/21
Valor de salida: 1
```


Ejercicio 4 Programa Java que lee una cadena desde la entrada estándar y la visualiza.

```
package EjemploProcesos;

import java.io.*;
public class Ejercicio4{
    public static void main (String [] args) {

        InputStreamReader in = new InputStreamReader(System.in);
        BufferedReader br= new BufferedReader(in);
        String texto;
        try {
            System.out.println("Introduce una cadena ...");
            texto = br.readLine();
            System.out.println("Cadena escrita: "+texto);
            in.close();
        } catch (Exception e) { e.printStackTrace(); }
    }
}
```

**Ejercicio 5: En linux, ejecutar los siguientes comandos:**

1. Crear el directorio “carpetita” en /alumno/Documentos
2. listar el directorio alumno
3. Hacer ping a “google.es”

```
package EjemplosProcesos;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;

public class LanzadorComandosJava {
    public printOutput getStreamWrapper(InputStream is, String type) {
        return new printOutput(is, type);
    }

    public static void main(String[] args) {

        Runtime rt = Runtime.getRuntime();
        LanzadorComandosJava rte = new LanzadorComandosJava();
    }
}
```



```
printOutput errorReported, outputMessage;

try {
    Process proc = rt.exec("mkdir /home/juan/carpetita");
    errorReported = rte.getStreamWrapper(proc.getErrorStream(), "ERROR");
    outputMessage = rte.getStreamWrapper(proc.getInputStream(), "OUTPUT");
    errorReported.start();
    outputMessage.start();
} catch (IOException e) {
    e.printStackTrace();
}

try {
    Process proc = rt.exec("ls -ltra /home/juan");
    errorReported = rte.getStreamWrapper(proc.getErrorStream(), "ERROR");
    outputMessage = rte.getStreamWrapper(proc.getInputStream(), "OUTPUT");
    errorReported.start();
    outputMessage.start();
} catch (IOException e) {
    e.printStackTrace();
}

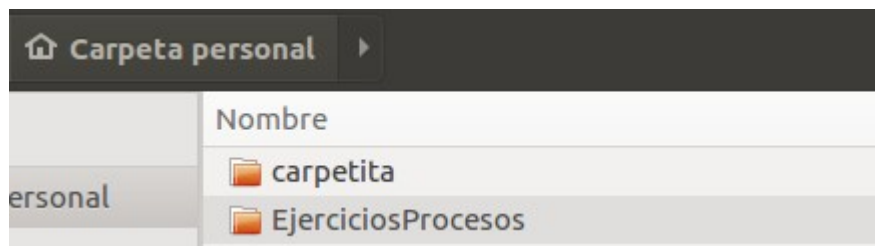
try {
    Process proc = rt.exec("ping google.es");
    errorReported = rte.getStreamWrapper(proc.getErrorStream(), "ERROR");
    outputMessage = rte.getStreamWrapper(proc.getInputStream(), "OUTPUT");
    errorReported.start();
    outputMessage.start();
} catch (IOException e) {
    e.printStackTrace();
}

}

private class printOutput extends Thread {
    InputStream is = null;

    printOutput(InputStream is, String type) {
        this.is = is;
    }

    public void run() {
        String s = null;
        try {
            BufferedReader br = new BufferedReader(
                new InputStreamReader(is));
            while ((s = br.readLine()) != null) {
                System.out.println(s);
            }
        } catch (IOException ioe) {
            ioe.printStackTrace();
        }
    }
}
}
```



```

Problemas @ Javadoc Declaración Consola
LanzadorComandosJava [Aplicación Java] /home/juan/.p2/pool/plugins/org.eclipse.justj.openjdk.h
total 2139324
-rw-r--r-- 1 juan juan      655 oct  6  2017 .profile
-rw-r--r-- 1 juan juan    3771 oct  6  2017 .bashrc
-rw-r--r-- 1 juan juan     220 oct  6  2017 .bash_logout
drwxr-xr-x 3 root root    4096 oct  6  2017 ..
drwxr-xr-x 2 juan juan    4096 oct  6  2017 Vídeos
drwxr-xr-x 2 juan juan    4096 oct  6  2017 Público
drwxr-xr-x 2 juan juan    4096 oct  6  2017 Música
drwx----- 3 juan juan    4096 oct  6  2017 .local
-rw-r--r-- 1 juan juan         0 oct  6  2017 .sudo as admin successful

drwxr-xr-x 56 juan juan    4096 oct 14 13:33 .
PING google.es (142.250.184.163) 56(84) bytes of data.
64 bytes from mad07s23-in-f3.1e100.net (142.250.184.163): icmp_seq=1 ttl=119 time=8.3
64 bytes from mad07s23-in-f3.1e100.net (142.250.184.163): icmp_seq=2 ttl=119 time=8.2
64 bytes from mad07s23-in-f3.1e100.net (142.250.184.163): icmp_seq=3 ttl=119 time=8.0
64 bytes from mad07s23-in-f3.1e100.net (142.250.184.163): icmp_seq=4 ttl=119 time=7.8
64 bytes from mad07s23-in-f3.1e100.net (142.250.184.163): icmp_seq=5 ttl=119 time=8.0
64 bytes from mad07s23-in-f3.1e100.net (142.250.184.163): icmp_seq=6 ttl=119 time=7.9

```

Clase ProcessBuilder

Esta clase de Java, crea y ejecuta procesos.

La clase que representa un proceso en Java es la clase Process. Los métodos ProcessBuilder.start() y Runtime.exec() crean un proceso nativo en el sistema operativo subyacente donde se está ejecutando la máquina virtual java y devuelve un objeto de la clase Process que pueden ser utilizado para controlar dicho proceso.

Métodos interesantes de la clase ProcessBuilder

- **enviroment():** devuelve las variables de entorno del proceso.
- **command():** devuelve los argumentos del proceso definido en test.
- **command(-con parametros-):** se define un nuevo proceso y sus argumentos
- **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 ()**, ejecutandose 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 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 maquina virtual java. 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 maquina virtual java. En este sentido , puede 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.

Ejemplos de ProcessBuilder

Ejemplo 1 (En windows)

```
package EjemplosProcesos;

import java.io.*;
import java.util.*;
public class EjemploProcessBuilder1 {
    public static void main (String args[]) {
        ProcessBuilder test = new ProcessBuilder();
        Map entorno = test.environment();
        System.out.println("Las variables de entorno son:");
        System.out.println(entorno);

        // Lanzamos el programa de Java Unsaludo
        test = new ProcessBuilder("java", "Unsaludo", "\"Hola Mundo!!\"");

        // 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());
        // en Windows 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 de la salida estándar
                System.out.println(linea);
        }
    }
}
```

```

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

```

Ejemplo: Creación de un proceso usando ProcessBuilder

```

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

```

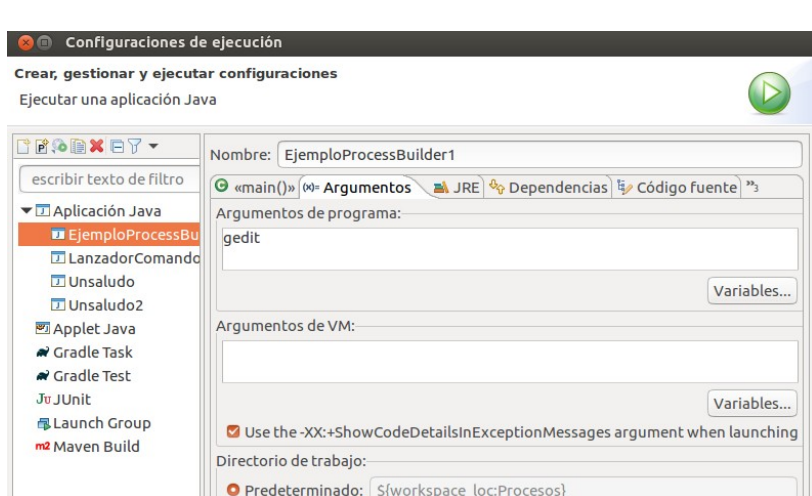
```

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);
        }
    }
}

```



1.5.2.2.- Terminación de procesos (operación **destroy**)

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.

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);
        }
    }
}
```

1.6.- Comunicación entre procesos

Un proceso es un programa en ejecución por lo que 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. 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 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 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.

En la mayoría de los sistemas operativos, estas entradas y salidas en procesos hijo son una copia de las mismas entradas y salidas que tuviera su padre. De tal forma que si se llama a la operación create dentro de un proceso que 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 streams¹:

- **OutputStream:** flujo de salida del proceso hijo. El *stream* está conectado por un **pipe**² 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 (*stdout*) del proceso hijo. Sin embargo, hay que saber que, por defecto, para la Máquina Virtual de Java, *stderr* está conectado al mismo sitio que *stdout*.

Si deseamos tenerlos separados para poder identificar los errores de forma más simple, 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 máquina virtual de Java serán diferentes y representarán la salida estándar y la salida de error del proceso de forma correspondiente.

Usando estos streams, el proceso padre puede enviarle datos al proceso hijo y recibir los resultados de salida que este genere comprobando los errores.

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 utilizado los stream visto anteriormente.

1 En Java a los flujos se los denominan como "STREAM" para lograr la comunicación entre el medio externo y nuestro programa (para saber si tiene que tratar con el monitor, el teclado, un socket o un sistema de ficheros); se acceden a las entradas y salidas estándar a través de campos estáticos de la clase java.lang.System; para cada flujo tenemos:

- System.in (entrada estándar)
- System.out (salida estándar)
- System.err (salida de error)

2 Una tubería (pipe, cauce o '|') consiste en una cadena de procesos conectados de forma tal que la salida de cada elemento de la cadena es la entrada del próximo. Permiten la comunicación y sincronización entre procesos. Es común el uso de buffer de datos entre elementos consecutivos.

Ejemplo de comunicación 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 ComunicacionBetweenProceso {

    public static void main (String args[] ) throws IOException{

        Process proceso = new ProcessBuilder(args).start();
        InputStream is = proceso.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);
        }

    }
}
```

Nota: Linux , MAC OS, Android, UNIX , IOS, etc. utilizan el formato de codificación de caracteres UTF-8, que utiliza por debajo el estandar UNICODE . Unicode es un estándar de codificación de caracteres diseñado para facilitar el tratamiento informático , transmisión y visualización de textos de múltiples lenguajes y disciplinas técnicas. El término “Unicode” viene de los tres objetivos perseguidos: universalidad, uniformidad y unicidad.

Diferentes versiones de Windows no utilizan UTF-8 , sino sus propios formatos de codificación no compatibles con el resto.

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.
- Usando JNI (Java Native Interface) . La utilización de Java permite abstraerse del comportamiento final de los procesos en los diferentes sistemas operativos.
- Librerías de comunicación no estándares entre procesos en Java que permiten aumentar las capacidades del estándar Java para comunicarnos. 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 (Java Native Interface) para poder utilizar métodos más cercanos al sistema operativo:
 - **Memoria compartida:** se establece una región de memoria compartida entre varios procesos a la cual pueden acceder todos ellos. Los procesos se comunican escribiendo y leyendo datos en ese espacio de memoria compartida.
 - **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.

1.7.-Redirección de la salida estándar. (En windows)

Los métodos:

- **redirectOutput()**
- **redirectError()**

```
package EjemploProcesos;
import java.io.File;
import java.io.IOException;
import java.lang.ProcessBuilder.Redirect;
public class Redireccion {
    public static void main(String[] args) {
        ProcessBuilder pb = new ProcessBuilder("C:/temp/miprograma.bat", "juanjo");
        File log = new File("C:/temp/log.txt");
        pb.redirectErrorStream(true);
        pb.redirectOutput(Redirect.appendTo(log));
        try {
            Process p = pb.start();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Donde **miprograma.bat** contiene:

```
echo Hola %1
```

Cuando ejecutamos el programa el fichero log.txt tendrá el siguiente contenido:



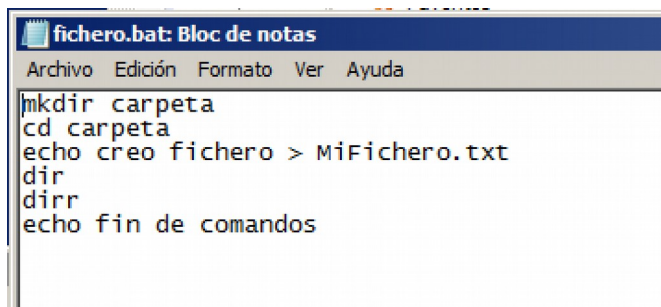
Podemos incluso ejecutar varios programas del S.O. dentro de un fichero bat.

```
package EjemploProcesos;
import java.io.File;
import java.io.IOException;
public class Redireccion {
    public static void main(String args[]) throws IOException {
        ProcessBuilder pb = new ProcessBuilder("CMD");
        File fBat = new File("c:/temp/fichero.bat");
        File fOut = new File("c:/temp/salida.txt");
        File fErr = new File("c:/temp/error.txt");

        pb.redirectInput(fBat);
        pb.redirectOutput(fOut);
        pb.redirectError(fErr);
    }
}
```

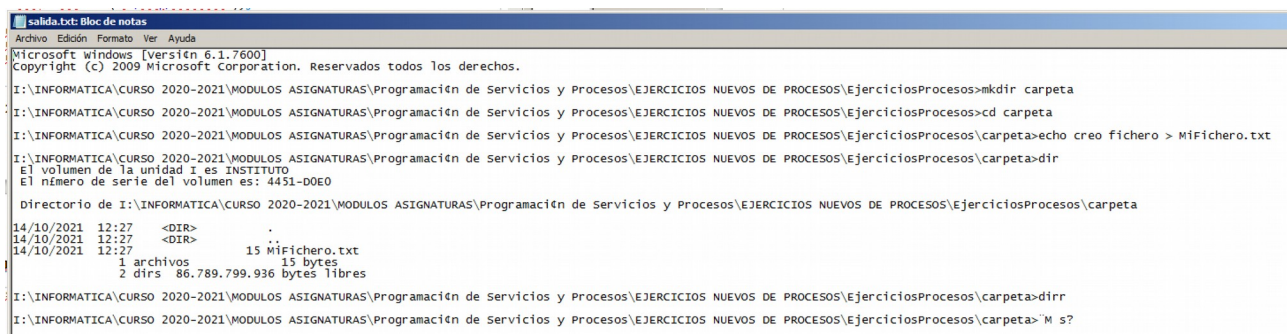
```
    pb.start();  
  }  
}
```

Contenido del fichero “fichero.bat”



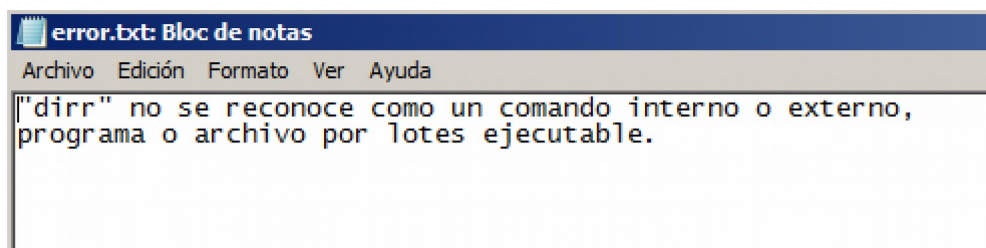
```
mkdir carpeta  
cd carpeta  
echo creo fichero > MiFichero.txt  
dir  
dirr  
echo fin de comandos
```

Contenido del fichero salida.txt



```
Microsoft Windows [Versión 6.1.7600]  
Copyright (c) 2009 Microsoft Corporation. Reservados todos los derechos.  
  
I:\INFORMATICA\CURSO 2020-2021\MODULOS ASIGNATURAS\Programación de Servicios y Procesos\EJERCICIOS NUEVOS DE PROCESOS\EjerciciosProcesos>mkdir carpeta  
I:\INFORMATICA\CURSO 2020-2021\MODULOS ASIGNATURAS\Programación de Servicios y Procesos\EJERCICIOS NUEVOS DE PROCESOS\EjerciciosProcesos>cd carpeta  
I:\INFORMATICA\CURSO 2020-2021\MODULOS ASIGNATURAS\Programación de Servicios y Procesos\EJERCICIOS NUEVOS DE PROCESOS\EjerciciosProcesos\carpeta>echo creo fichero > MiFichero.txt  
I:\INFORMATICA\CURSO 2020-2021\MODULOS ASIGNATURAS\Programación de Servicios y Procesos\EJERCICIOS NUEVOS DE PROCESOS\EjerciciosProcesos\carpeta>dir  
El volumen de la unidad I es INSTITUTO  
El número de serie del volumen es: 4451-D0E0  
  
Directorio de I:\INFORMATICA\CURSO 2020-2021\MODULOS ASIGNATURAS\Programación de Servicios y Procesos\EJERCICIOS NUEVOS DE PROCESOS\EjerciciosProcesos\carpeta  
14/10/2021 12:27 <DIR> .  
14/10/2021 12:27 <DIR> ..  
14/10/2021 12:27      15 MiFichero.txt  
                1 archivos      15 bytes  
                2 dirs  86.789.799.936 bytes libres  
I:\INFORMATICA\CURSO 2020-2021\MODULOS ASIGNATURAS\Programación de Servicios y Procesos\EJERCICIOS NUEVOS DE PROCESOS\EjerciciosProcesos\carpeta>dirr  
I:\INFORMATICA\CURSO 2020-2021\MODULOS ASIGNATURAS\Programación de Servicios y Procesos\EJERCICIOS NUEVOS DE PROCESOS\EjerciciosProcesos\carpeta>"M S?"
```

Contenido del fichero error.txt



```
"dirr" no se reconoce como un comando interno o externo,  
programa o archivo por lotes ejecutable.
```

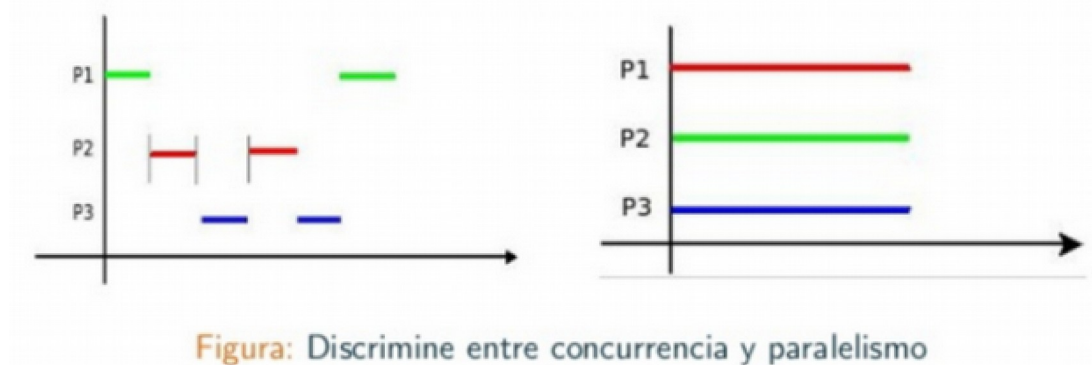
2.- Programación concurrente

Concurrente = concurso de varios procesos en un mismo *tiempo*.

Un programa **en ejecución** puede dar lugar a uno o más procesos.

Dos procesos son concurrentes cuando existe un intercalado o solapamiento en el tiempo. Si la ejecución es simultánea estaremos hablando de programación paralela.

Entonces podemos hablar de proceso como una **actividad asíncrona susceptible** de ser asignada a un procesador.



2.1.- Beneficios de la programación concurrente.

- **Mejor aprovechamiento de la CPU.** un proceso no ocupa CPU si está en una operación de E/S.
- **Velocidad de ejecución.** Se pueden repartir procesos entre distintos procesadores o gestionar en un único procesador según su importancia.

Solución a problemas de naturaleza concurrente.

- **Sistemas de control.** Se capturan datos a través de sensores se analizan y actúan en función del análisis. Por ejemplo sistemas en tiempo real.
- **Tecnologías web.** Un servidor web o de correo electrónico atiende múltiples peticiones de los usuarios concurrentemente.
- **Aplicaciones basadas en GUI.** Por ejemplo un navegador está navegando por una página mientras está descargando un archivo.
- **Simulación.** Programas que modelan sistemas físicos con autonomía.
- **Sistemas gestores de BBDD.** Cada usuario puede ser visto como un proceso.

2.2.-Concurrencia y hardware.

En un sistema **monoprocesador**, el S.O. va alternando el tiempo entre los distintos procesos.

Se denomina **multiproceso** a la gestión de varios procesos dentro de un sistema **multiprocesador**, donde cada procesador puede acceder a una memoria común.

2.3.- Programas concurrentes.

- **Dos o más procesos decimos que son concurrentes, paralelos, o que se ejecutan concurrentemente,** cuando son procesados al mismo tiempo, es decir, que para ejecutar uno de ellos, no hace falta que se haya ejecutado otro.

- **En sistemas multiprocesador**, esta ejecución simultánea podría conseguirse completamente, puesto que podremos asignarle, por ejemplo, un proceso A al procesador A y un proceso B al procesador B y cada procesador realizaran la ejecución de su proceso.
- **Cuando tenemos un solo procesador** se producirá un intercalado de las instrucciones de ambos procesos, de tal forma que tendremos la sensación de que hay un paralelismo en el sistema (conurrencia, ejecución simultánea de más de un proceso).
- *Ahora bien, está claro que en esto tenemos que tener en cuenta que mientras un proceso está escribiendo un valor en una variable determinada, puede darse el caso que otro proceso que es concurrente al primero vaya a leer o escribir en esa misma variable, entonces habrá que estudiar el caso en el que un proceso haga una operación sobre una variable (o recurso en general) y otro proceso concurrente a él realice otra operación de tal forma que no se realice correctamente. Para estudiar esto, y determinar el tipo de operaciones que se pueden realizar sobre recursos compartidos utilizaremos las condiciones de Bernstein.*

3.- Programación paralela y distribuida

3.1.- Programación paralela

Un **programa paralelo** es un programa concurrente diseñado para ejecutarse en un mismo multiprocesador. Permite que muchos elementos de proceso independientes (por ejemplo muchos ordenadores conectados en red, un equipo con varios procesadores, o una combinación de ambos) trabajen simultáneamente para resolver un problema. El problema a resolver se divide en partes INDEPENDIENTES, de forma que se pueda ejecutar a la vez que las demás partes.

La comunicación entre procesos puede realizarse por:

- **Memoria compartida.** Cuando los micros comparten físicamente la memoria.
- **Paso de mensajes.** Cuando cada procesador dispone de su propia memoria independiente. En este caso cada procesador pide información a otros procesadores mediante un entorno que permita hacerlo.

El **paralelismo** se utiliza normalmente en centros de supercomputación, pero con la aparición de procesadores de múltiples núcleos su utilización ya es posible trasladarla a equipos más asequibles.

Ventajas e inconvenientes

Ventajas del procesamiento paralelo:

- Ejecución simultánea de tareas.
- Disminuye el tiempo total de ejecución
- Resuelve problemas complejos y de grandes dimensiones.
- Utilización de recursos no locales distribuidos en la red

- Disminución de costos, aprovechando los recursos distribuidos, no es necesario gastar en un único supercomputador, se puede alcanzar el mismo poder de computación con equipos más modestos distribuidos.

Inconvenientes:

- Los compiladores y entornos de programación para sistemas paralelos son más complicados de desarrollar.
- Los programas paralelos son más difíciles de escribir
- Hay mayor consumo de energía
- Mayor complejidad en el acceso a datos
- Complejidad a la hora de la comunicación y sincronización de las diferentes subtareas.

Ejemplos de utilización de la programación paralela

- Estudios meteorológicos
- Estudios del genoma humano
- Modelado de la biosfera
- Predicciones sísmicas
- Simulación de moléculas

Ejemplos de programación paralela

- Búsqueda de inteligencia extraterrestre: https://setiathome.berkeley.edu/sah_about.php

3.2.- Programación distribuida

Nace de la necesidad de compartir recursos. El sistema distribuido más grande es sin duda el propio Internet (web, eMail, ficheros, videoconferencias, mensajería, ...). El Cloud Computing o servicios de computación en la nube es una de las aplicaciones más recientes. Un sistema es distribuido cuando los componentes software están distribuidos en la red, se comunican y coordinan mediante el paso de mensajes. Esta definición tiene las siguientes consecuencias:

- Concurrencia: ejecución de programas concurrentes.
- Inexistencia de un reloj global. Implica sincronizarse con el paso de mensajes.
- Fallos independientes: cada componente del sistema puede fallar sin que perjudique la ejecución de los demás.

La programación distribuida es un paradigma de programación enfocado en desarrollar:

- Sistemas distribuidos
- Abiertos
- escalables
- Transparentes
- Tolerante a fallos

Una arquitectura típica de sistema distribuido es la del cliente-servidor, el cliente es la parte activa el que lleva la iniciativa del programa y los servidores la parte pasiva que realiza tareas bajo requerimiento de los clientes.

Comunicación entre procesos en un sistema distribuido

- **Sockets** (IP+puerto): son de muy bajo nivel de abstracción.
- **RPC** (Remote Procedure Call): permite a un cliente llamar a un procedimiento remoto que está en ejecución en un proceso servidor. El proceso define en su interfaz de servicio que procedimientos están disponibles para ser llamados remotamente.
- **Invocación remota de objetos:** es una extensión de la programación basada en objetos, donde los procesos invocan a un método remoto o RMI (Remote Method Invocation). Es decir un objeto de un proceso puede invocar a un método de otro objeto que está en otro proceso.

Java RMI para ver como soporta Java los objetos distribuidos. RMI se caracteriza por la facilidad de su uso en la programación por estar específicamente diseñado para Java; proporciona paso de objetos por referencia, recolección de basura distribuida (Garbage Collector distribuido) y paso de tipos arbitrarios.

Toda aplicación RMI normalmente se descompone en 2 partes:

- **Un servidor**, que crea algunos objetos remotos, crea referencias para hacerlos accesibles, y espera a que el cliente los invoque.
- **Un cliente**, que obtiene una referencia a objetos remotos en el servidor, y los invoca.

Un servidor RMI consiste en definir un objeto remoto que va a ser utilizado por los clientes. Para crear un objeto remoto, se define una interfaz, y el objeto remoto será una clase que implemente dicha interfaz. Veamos como crear un servidor de ejemplo mediante 3 pasos:

- Definir la interfaz remota. Cuando se crea una interfaz remota:
 - La interfaz debe ser pública.
 - Debe heredar de la interfaz **java.rmi.Remote**, para indicar que puede llamarse desde cualquier máquina virtual Java.
 - Cada método remoto debe lanzar la excepción **java.rmi.RemoteException** en su cláusula throws, además de las excepciones que pueda manejar.

Ventajas de la programación distribuida

- Se comparten recursos y datos
- Crecimiento incremental
- Mayor flexibilidad para distribuir la carga
- Alta disponibilidad
- Soporte de aplicaciones distribuidas

- Filosofía abierta y heterogénea

Inconvenientes de la programación distribuida

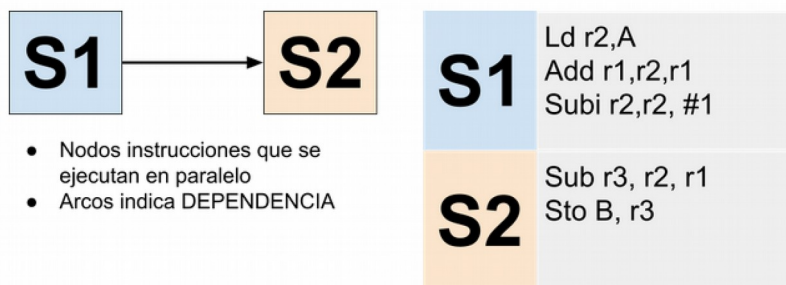
- Aumenta la complejidad
- Se necesita software nuevo especializado
- Problemas derivados de las comunicaciones (perdidas, saturaciones, etc.)
- Problemas de seguridad, ataques DDoS

4.- Condiciones de Bernstein

4.1.-Condiciones para el paralelismo: DEPENDENCIAS.

- Para poder ejecutar SEGMENTOS DE CÓDIGO en paralelo, hay que asegurarnos que son INDEPENDIENTES, es decir el orden de ejecución no afecta al resultado.
- Las dependencias entre grupos de instrucciones pueden describirse por medio del GRAFO DE DEPENDENCIAS.
- El análisis de dependencias es estático, es decir, no se realiza en tiempo de ejecución (run-time).

GRAFO DE DEPENDENCIAS

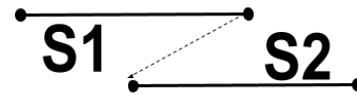
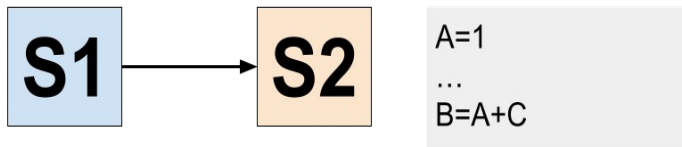


Existen 3 tipos de dependencias:

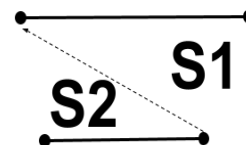
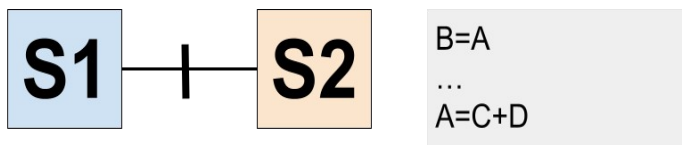
- (1) **De datos**, cuando 2 instrucciones hacen uso del mismo dato el orden de ejecución es importante:
 - Dependencias de flujo
 - Antidependencias
 - Dependencias de salida
 - Dependencias de E/S (fichero)
 - Desconocidas. Ciertas dependencias que no pueden ser determinadas, direccionamientos indirectos, cuando las variables de índices aparecen más de una vez

en un bucle con índices diferentes, índices no lineales, índices que no continenen la variable del bucle.

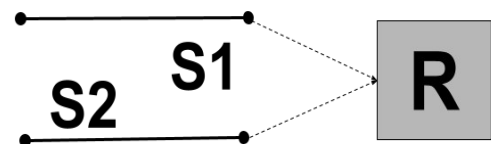
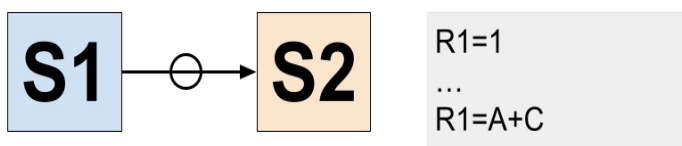
- **Dependencia de flujo:** la salida de S1 alimenta la entrada de S2



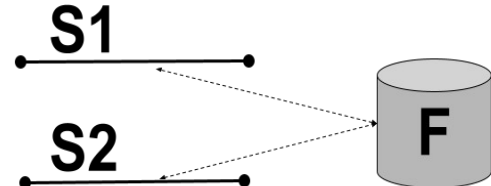
- **Antidependencia:** la salida de S2 se solapa con la entrada de S1



- **Dependencia de salida:** S1 y S2 escriben en la misma salida



- **Dependencia de I/O:** S1 y S2 acceden al mismo fichero



Las dependencias de E/S a ficheros y las desconocidas presuponen el peor caso posible, por lo que siempre se asumirá una dependencia para evitar posibles errores.

(2) Dependencias de control

- Cuando el orden de ejecución no puede ser determinado estáticamente, sino sólo en tiempo de ejecución.
- Ejemplo: sentencias tipo IF, branch/jump.
- También aparecen en bucles (ver ejemplo).
- Dado que el orden no es conocido anticipadamente, las dependencias reales no pueden ser determinadas.

- (3) Dependencias de recursos

- Tenemos dependencias de recursos cuando recursos compartidos del procesador (ALU, buses, etc.) son demandados por instrucciones diferentes al mismo tiempo.

5.- 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.

Los envíos y recepciones por los flujos de datos permiten a un proceso hijo comunicarse con su padre a través de un canal de comunicación unidireccional bloqueante.

5.1.- Espera de procesos (operación wait)

Además de la utilización de los flujos de datos , también se puede esperar por la finalización del proceso hijo y obtener su información de finalización mediante la operación **wait** . Dicha operación bloquea al proceso padre hasta que el hijo finaliza su ejecución mediante **exit** . 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 .

Se puede utilizar **exitValue ()** para obtener valores de retorno que devolvió un proceso hijo.

Implementación de sincronización de procesos

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

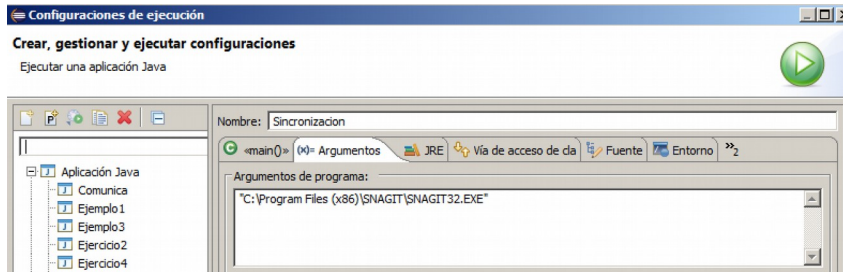
public class ProcessSincronization {

    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)
                    + "devolvio: " + retorno);
        }
        catch (IOException e){
            System.out.println ("Error ocurrió ejecutandose el
                comando . Descripción : "+ e.getMessage());
        }
        catch (InterruptedException e){
            System.out.println ("El comando fue interrumpido.
                Descripción del error:" + e.getMessage());
        }
    }
}
```

```
}  
}
```

Buscamos la ruta de alguna aplicación que podamos abrir .



5.2.- Programación 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 . La multiprogramación puede producirse entre procesos totalmente independientes, como podrían ser los correspondientes al procesador de textos, navegador, reproductor de música, etc. , o entre procesos que pueden cooperar entre sí para realizar una tarea común.

Si se pretende realizar procesos que cooperen entre sí, debe ser el desarrollador quien lo implemente utilizando la comunicación y sincronización de procesos.

A la hora de realizar un programa multiproceso cooperativo, se deben seguir las siguientes fases:

- Descomposición funcional: Es necesario identificar previamente las diferentes funciones que debe realizar la aplicación y las relaciones entre ellas.
- Partición: Distribución de las diferentes funciones en procesos estableciendo el esquema de comunicación entre los mismos. Al ser procesos cooperativos necesitarán información unos de otros por lo que deben comunicarse. (objetivo maximizar la independencia entre los procesos minimizando la comunicación entre ellos).
- Implementación: Una vez realizada la descomposición y la partición se realiza la implementación utilizando las herramientas disponible por la plataforma a utilizar. En este caso, Java permite únicamente métodos sencillos de comunicación y sincronización de procesos para realizar la cooperación.

5.3.- Clase Process

Para realizar algoritmos multiprocesos en Java, usaremos la clase Process. Algunos métodos son:

Método	Tipo de retorno	Descripción
<i>getOutputStream()</i>	<i>OutputStream</i>	Obtiene el flujo de salida del proceso hijo conectado al <i>stdin</i>
<i>getInputStream()</i>	<i>InputStream</i>	Obtiene el flujo de entrada del proceso hijo conectado al <i>stdout</i> del proceso hijo
<i>getErrorStream()</i>	<i>InputStream</i>	Obtiene el flujo de entrada del proceso hijo conectado al <i>stderr</i> del proceso hijo
<i>destroy()</i>	<i>void</i>	Implementa la operación <i>destroy</i>
<i>waitFor()</i>	<i>int</i>	Implementa la operación <i>wait</i>
<i>exitValue()</i>	<i>int</i>	Obtiene el valor de retorno del proceso hijo