UNIDAD1 - PROGRAMACIÓN MULTIPROCESO

Programación de Servicios y Procesos

Profesora: Raquel Barreales Quintanilla

UNIDAD 1 - PROGRAMACIÓN MULTIPROCESO

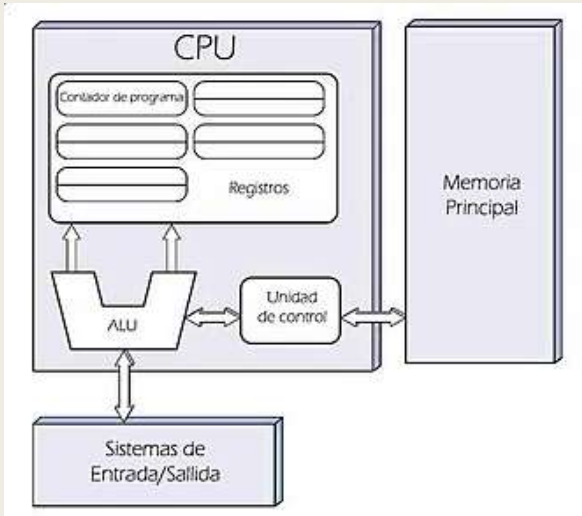
- Programas y procesos
- Procesos y Servicios
- Estados de un proceso
- Gestión de procesos en Linux
- Creación de procesos en Java
- Programación concurrente
- Programación paralela y distribuida

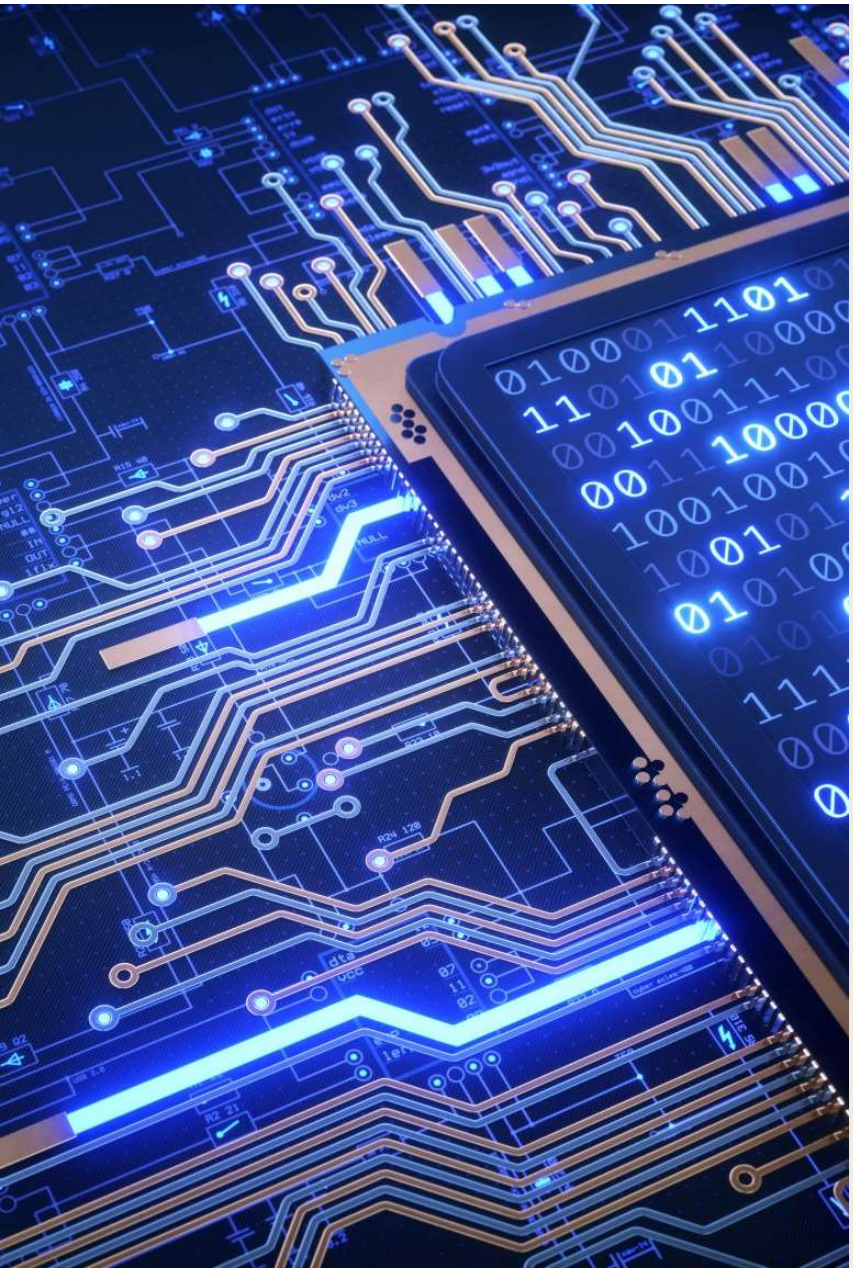
Programas y procesos

- Un **programa** contiene un conjunto de instrucciones que se pueden ejecutar directamente en una máquina. Esto puede ser, una máquina física o una máquina virtual (java).
- Un programa es un objeto estático almacenado en un medio
- Un **proceso** corresponde a una instancia de un programa en ejecución. (CARGADO EN MEMORIA)
- Un proceso es una entidad dinámica que en su ejecución puede crear más procesos.

Programas y procesos

- John von Neumann
- CPU(Central Processing Unit)
  - *Ejecuta el proceso*
  - *Datos cargados en procesador*
- MEMORIA – CELDAS
  - *Asignada al proceso y/o dinámicamente*
- E/S





Programas y procesos

- Toda la información asociada a un proceso se guarda en el bloque de control de procesos(PCB o process control block)
- Si un programa se ejecuta varias veces → varios procesos y PCB
- Jerarquía de procesos → pstree
- Uso de memoria y procesador →top
- Para descargar el proces explorer: https://learn.microsoft.com/es-es/sysinternals/downloads/process-explorer

Multitarea(multitasking)

- Ejecución simultanea de más de un proceso en un procesador a lo largo de un intermedio de tiempo.
- Ejecución de proceso **sin multitarea**

CPU	A	A	A	A	A	B	B	B	B	C	C	C	C	C	C
Tiempo	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Multitarea(multitasking)

- Ejecución de proceso **CON MULTITAREA**

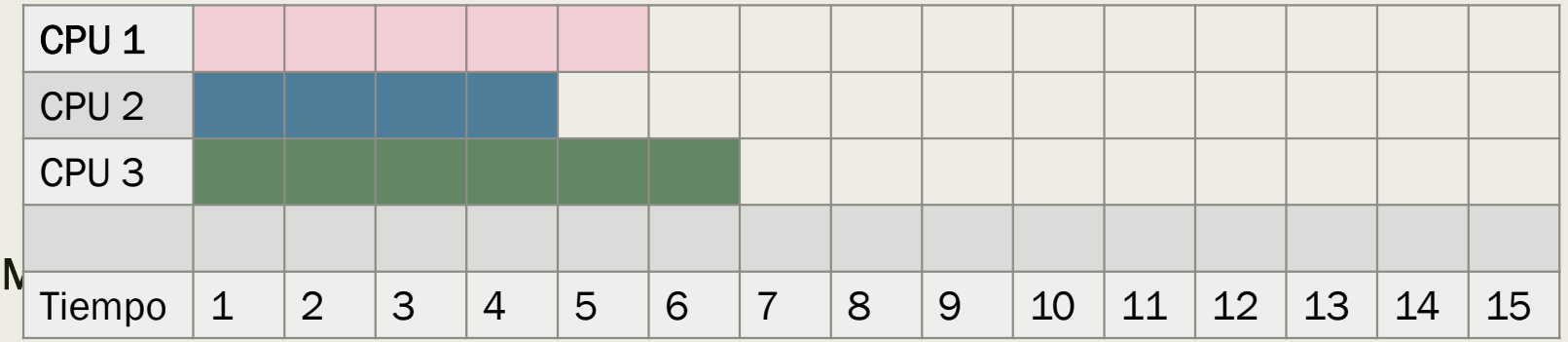
A	■			■			■			■			■		
B		■			■			■			■				
C			■			■			■			■		■	■
CPU	A	B	C	A	B	C	A	B	C	A	B	C	A	C	C
Tiempo	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Multitarea(multitasking)

- Antes de ejecutar un nuevo proceso, se guarda el estado del actual para retomar su ejecución
- Se guarda contenido del contador y registros del procesador y se cargan los del siguiente. (cambio de contexto)
- Sistema multiprocesador – ejecución en paralelo

Multitarea(multitasking)

- Ejecución de proceso **CON MULTIPROCESO**



- M

Multitarea(multitasking)

- La multitarea requiere tiempo extra para el cambio de contexto.
- Uso del procesador→ aprovechar tiempo en el que no está en uso
- Tiempo de operaciones E/S

E/S		A	A	A		A	A			A	A	A	A	A			A	A			
			B		B	B		B	B	B		B			B	B			B	B	
CPU	A	B		B	A		B	A	A		B		B	B	A	A	B	B	A	A	B
Tiempo	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21

Procesos y Servicios

- **Proceso**: Es un programa o aplicación en ejecución y bajo el control del sistema operativo.
- **Servicio**: Es un programa o aplicación en segundo plano, no interactúa con el usuario. Habitualmente, un servicio es un programa que atiende a otro programa.

Estados de un Proceso

Estados de un proceso: Un proceso puede atravesar diversas etapas en su «ciclo de vida». Los estados en los que puede estar son:

- ***En ejecución:*** *está dentro del microprocesador.*
- ***Pausado/detenido/en espera:*** *el proceso tiene que seguir en ejecución, pero en ese momento el S.O tomó la decisión de dejar paso a otro.*
- ***Interrumpido:*** *el proceso tiene que seguir en ejecución, pero el usuario ha decidido interrumpir la ejecución.*

Funcionamiento

- Cuando el sistema operativo decide parar un proceso y arrancar otro, se debe rearrancar en el mismo estado en el que se encontraba cuando se paró, toda esa información se almacena en el **Bloque de Control de Proceso.**

El Bloque de Control de Proceso (BCP)

- BCP **Bloque de Control de Proceso** estructura de datos donde se almacena la información de un proceso:
  - *Identificación del proceso*
  - *Estado del proceso*
  - *Contador de programa*
  - *Registros de CPU*
  - *Prioridad del proceso*
  - *Información de gestión de memoria*
  - *Tiempo de CPU y tiempo real consumido*
  - *Información de estado de E/S: lista de dispositivos asignados, archivos abiertos, etc.*

MULTIPROCESO

Ejecutar más de un proceso a la vez, varias CPU's.

Programación multiproceso o multitarea: 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, dividiéndose un proceso en varios subprocesos.

Gestión de procesos en Linux

ps (process status) - *Ver parte de la información asociada a cada proceso*

PID: identificador del proceso

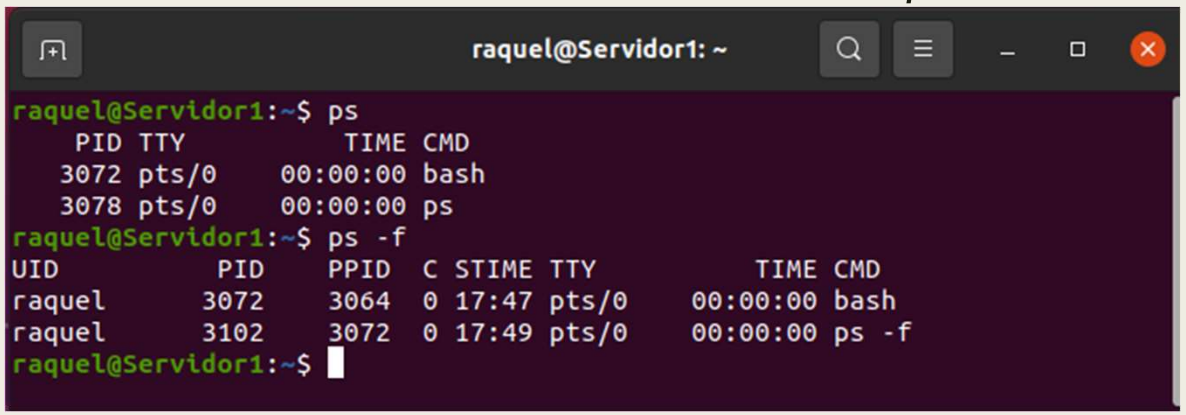
TTY: terminal asociado del que lee y escribe

TIME: cantidad total de tiempo de CPU que el proceso ha utilizado desde que nació

CMD: nombre del proceso

Gestión de procesos en Linux

ps -f : Muestra más información asociada a cada proceso



UID: nombre del usuario

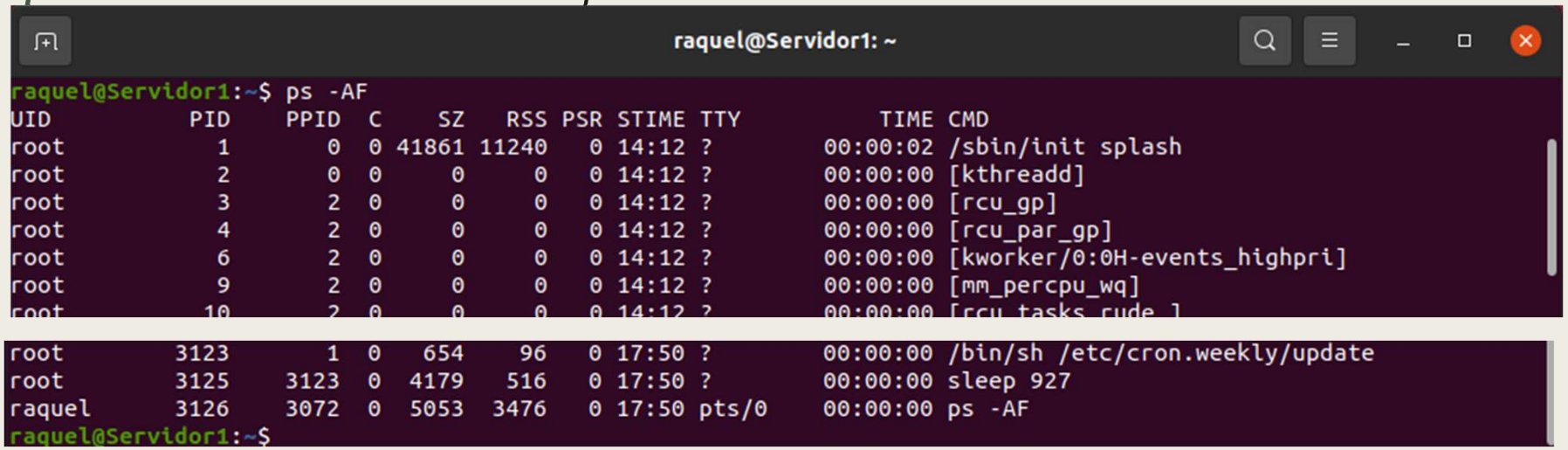
PPID: PID del padre de cada proceso

C: porcentaje de recursos de CPU utilizado por el proceso

STIME: hora de inicio del proceso

Gestión de procesos en Linux

ps -AF : Muestra todos los procesos activos con todos los detalles



SZ: Tamaño virtual de la imagen de un proceso. Esto incluye código, datos, librerías compartidas usadas por los procesos.

RSS: Tamaño de la parte residente en memoria en kilobytes

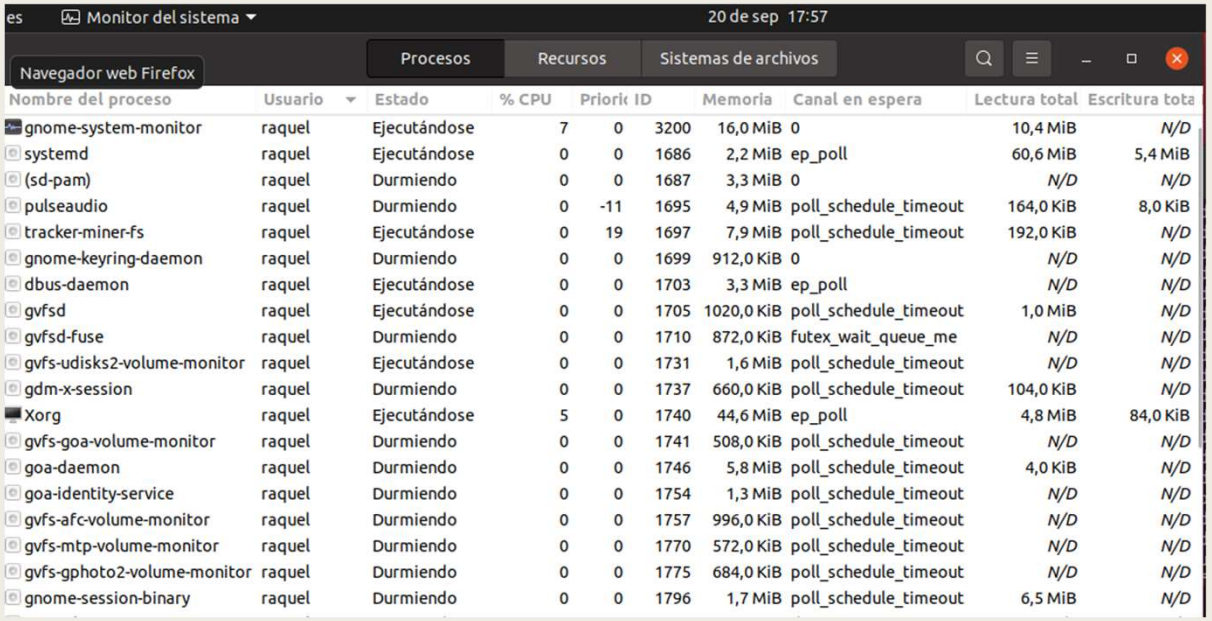
PSR: Procesador que el proceso tiene actualmente asignado

Gestión de procesos en Linux - Ventana

En Ubuntu desde:

Menú **Sistema -> Administración -> Monitor** del sistema

Podemos acceder a la interfaz gráfica que nos muestra información sobre los procesos que se están ejecutando.



Nombre del proceso	Usuario	Estado	% CPU	Priorid	ID	Memoria	Canal en espera	Lectura total	Escritura tota
gnome-system-monitor	raquel	Ejecutándose	7	0	3200	16,0 MiB	0	10,4 MiB	N/D
systemd	raquel	Ejecutándose	0	0	1686	2,2 MiB	ep_poll	60,6 MiB	5,4 MiB
(sd-pam)	raquel	Durmiendo	0	0	1687	3,3 MiB	0	N/D	N/D
pulseaudio	raquel	Durmiendo	0	-11	1695	4,9 MiB	poll_schedule_timeout	164,0 KiB	8,0 KiB
tracker-miner-fs	raquel	Ejecutándose	0	19	1697	7,9 MiB	poll_schedule_timeout	192,0 KiB	N/D
gnome-keyring-daemon	raquel	Durmiendo	0	0	1699	912,0 KiB	0	N/D	N/D
dbus-daemon	raquel	Ejecutándose	0	0	1703	3,3 MiB	ep_poll	N/D	N/D
gvfsd	raquel	Ejecutándose	0	0	1705	1020,0 KiB	poll_schedule_timeout	1,0 MiB	N/D
gvfsd-fuse	raquel	Durmiendo	0	0	1710	872,0 KiB	futex_wait_queue_me	N/D	N/D
gvfs-udisks2-volume-monitor	raquel	Ejecutándose	0	0	1731	1,6 MiB	poll_schedule_timeout	N/D	N/D
gdm-x-session	raquel	Durmiendo	0	0	1737	660,0 KiB	poll_schedule_timeout	104,0 KiB	N/D
Xorg	raquel	Ejecutándose	5	0	1740	44,6 MiB	ep_poll	4,8 MiB	84,0 KiB
gvfs-goa-volume-monitor	raquel	Durmiendo	0	0	1741	508,0 KiB	poll_schedule_timeout	N/D	N/D
goa-daemon	raquel	Durmiendo	0	0	1746	5,8 MiB	poll_schedule_timeout	4,0 KiB	N/D
goa-identity-service	raquel	Durmiendo	0	0	1754	1,3 MiB	poll_schedule_timeout	N/D	N/D
gvfs-afc-volume-monitor	raquel	Durmiendo	0	0	1757	996,0 KiB	poll_schedule_timeout	N/D	N/D
gvfs-mtp-volume-monitor	raquel	Durmiendo	0	0	1770	572,0 KiB	poll_schedule_timeout	N/D	N/D
gvfs-gphoto2-volume-monitor	raquel	Durmiendo	0	0	1775	684,0 KiB	poll_schedule_timeout	N/D	N/D
gnome-session-binary	raquel	Durmiendo	0	0	1796	1,7 MiB	poll_schedule_timeout	6,5 MiB	N/D

Procesos en Windows

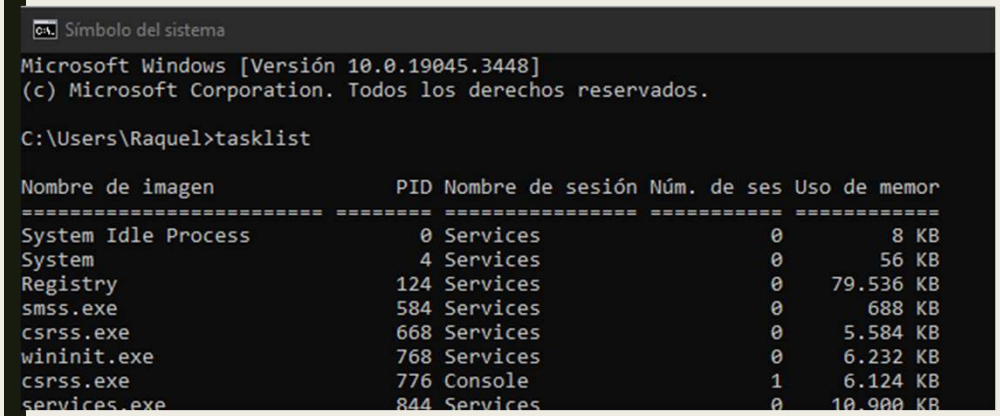
Desde la línea de comandos podemos utilizar el comando **tasklist** para ver los procesos que se están ejecutando

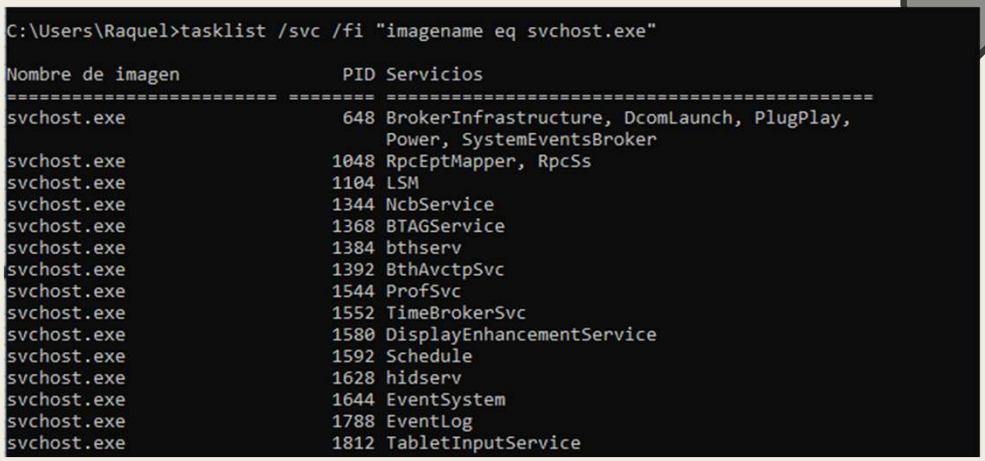
tasklist

Para ver los servicios que se están ejecutando bajo el proceso **svchost.exe** (Host de servicios, es un proceso de Windows que debe estar siempre abierto. Este servicio de Windows ejecuta bastantes procesos en segundo plano)

tasklist /svc /fi "imagename eq svchost.exe"

También vemos los procesos en el Administrador de tareas de Windows, en la pestaña Procesos

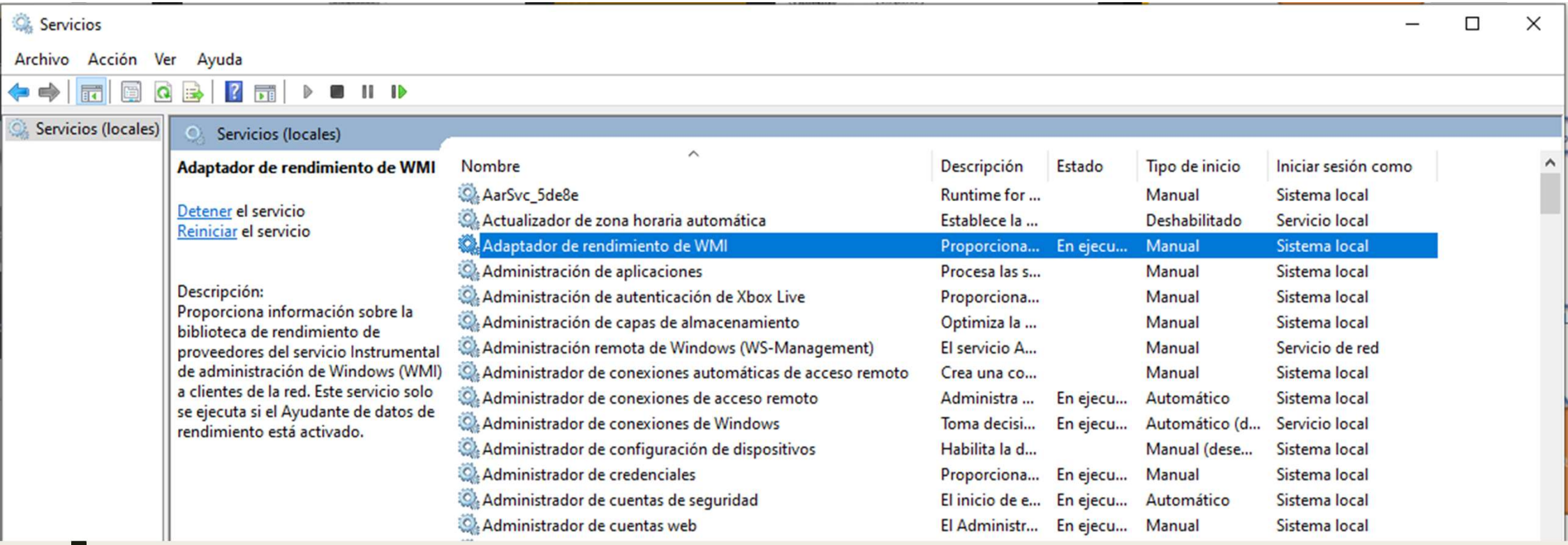




Procesos en Windows

tasklist

tasklist /svc /fi "imagename eq svchost.exe"



Programación

SERVICIOS

Servicios en windows→ permite arrancar y parar servicios

Servicios en Linux

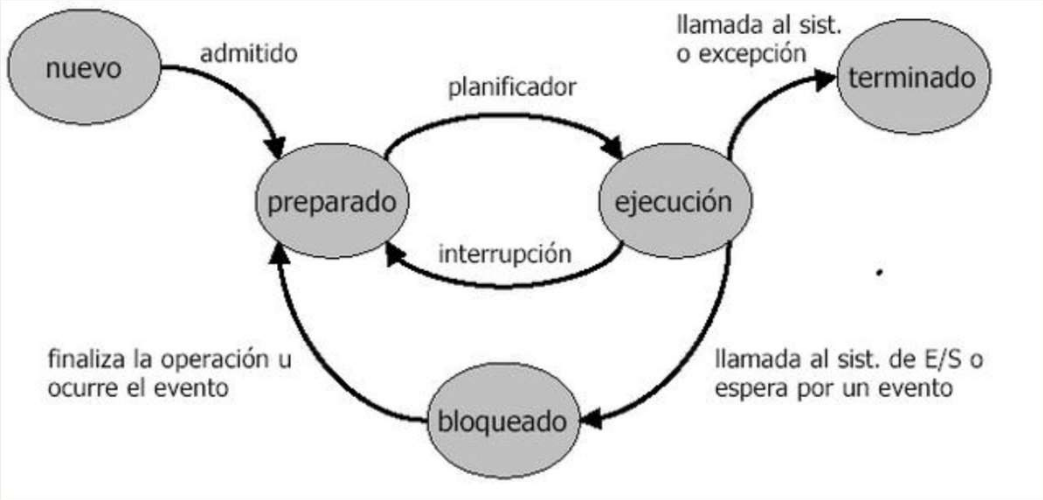
- Si aparece enable en la columna STATE se arranca al inicio
- systemctl list-unit-files

- Ver todos los servicios cargados en el sistema
- systemctl --type=service
- systemctl -t=service

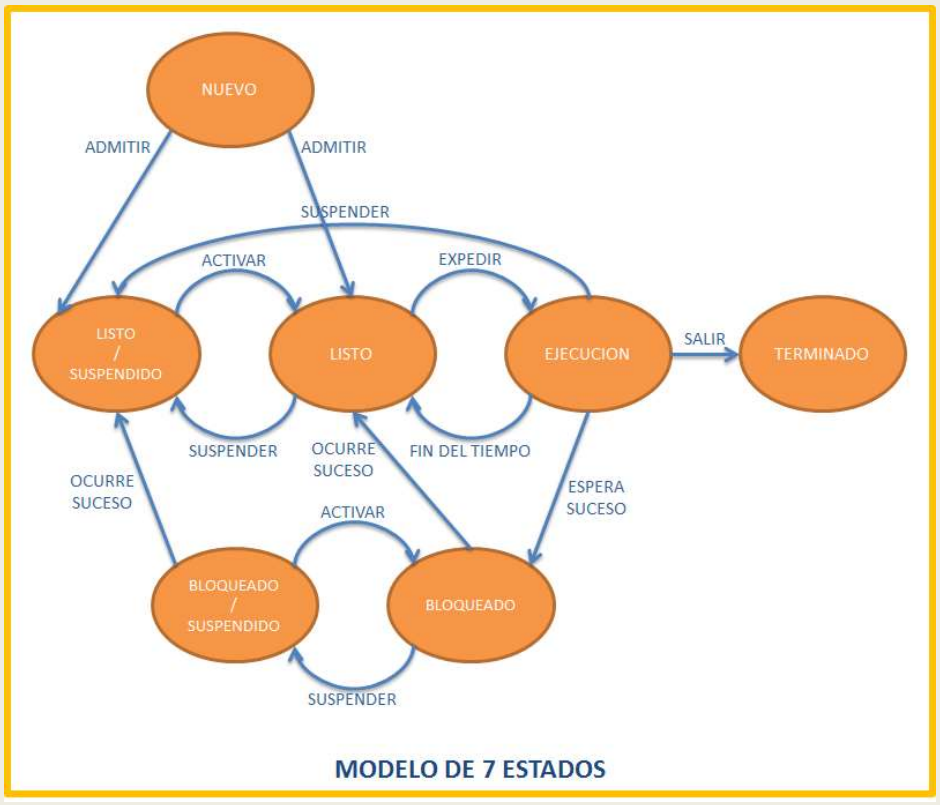
Servicios en Linux - acciones

Acción	Función
status	Muestra el estado de ejecución del servicio
start	Arranca el servicio
stop	Para el servicio
restart	Reinicia el servicio
reload	Reinicia con los cambios
enable	Establece el arranque automático al inicio
disable	Establece que no se arranque automáticamente

ESTADOS DE UN PROCESO



Modelo de 7 estados



ESTADOS DE UN PROCESO - EJEMPLO

Creamos un proceso sencillo en Linux: guardamos y le asignamos un nombre al script script1.sh

```
#!/bin/bash
while :
do
    echo presiona CTRL + C para parar
    echo presiona CTRL + Z para pausar
done
```

ESTADOS DE UN PROCESO - EJEMPLO

Ejecutamos el proceso **sh nombreScript**
Paramos su ejecución por tecleando **^Z (CRTL+Z)**

Para ver procesos detenidos
$ jobs
Matar procesos detenidos
$ kill %1
Reanudar proceso
$ fg 1

Matar procesos:
kill -9 pid . Matar el proceso

```
presiona CTRL + C para parar
presiona CTRL + Z para pausar
presiona CTRL + C para parar
presio^Z
[1]+  Detenido                 sh script1.sh
root@Servidor1:/home/raquel/Documentos/scripts# jobs
[1]+  Detenido                 sh script1.sh
root@Servidor1:/home/raquel/Documentos/scripts# ps -af
UID          PID    PPID  C STIME TTY          TIME CMD
raquel      1740    1737  0 14:13 tty2     00:00:51 /usr/lib/xorg/Xorg vt
raquel      1796    1737  0 14:13 tty2     00:00:00 /usr/libexec/gnome-se
root        3541    3072  0 18:09 pts/0    00:00:00 sudo su
root        3542    3541  0 18:09 pts/0    00:00:00 su
root        3543    3542  0 18:09 pts/0    00:00:00 bash
root        5720    3543  0 23:41 pts/0    00:00:01 sh script1.sh
root        5740    5687  0 23:43 pts/1    00:00:00 sudo su
root        5741    5740  0 23:43 pts/1    00:00:00 su
root        5742    5741  0 23:43 pts/1    00:00:00 bash
root        5758    3543  0 23:46 pts/0    00:00:00 ps -af
```

ESTADOS DE UN PROCESO - EJEMPLO

Ejecutamos el proceso **sh nombreScript**
Paramos su ejecución por tecleando **^Z (CRTL+Z)**

Para ver procesos detenidos
$ jobs
Matar procesos detenidos
$ kill %1
Reanudar proceso
$ fg 1

Matar procesos:
kill -9 pid . Matar el proceso

```
root        5758    3543  0 23:46 pts/0    00:00:00 ps -af
root@Servidor1:/home/raquel/Documentos/scripts# kill %1

[1]+  Detenido                 sh script1.sh
root@Servidor1:/home/raquel/Documentos/scripts# ps -af
UID          PID    PPID  C STIME TTY          TIME CMD
raquel      1740    1737  0 14:13 tty2     00:00:52 /usr/lib/xorg/Xorg vt
raquel      1796    1737  0 14:13 tty2     00:00:00 /usr/libexec/gnome-se
root        3541    3072  0 18:09 pts/0    00:00:00 sudo su
root        3542    3541  0 18:09 pts/0    00:00:00 su
root        3543    3542  0 18:09 pts/0    00:00:00 bash
root        5740    5687  0 23:43 pts/1    00:00:00 sudo su
root        5741    5740  0 23:43 pts/1    00:00:00 su
root        5742    5741  0 23:43 pts/1    00:00:00 bash
root        5767    3543  0 23:48 pts/0    00:00:00 ps -af
[1]+  Terminado                sh script1.sh
root@Servidor1:/home/raquel/Documentos/scripts#
```

Concurrencia JAVA

- La máquina virtual, Java y la biblioteca de clases estándar soportan concurrencia.
- La **clase Process** soporta la funcionalidad básica de procesos
- Es abstracta → no podemos crear objetos de ella
- Tiene métodos para lanzar un proceso, saber su estado o controlar su ejecución.
- Crearemos objetos con métodos de otras clases
  - *ProcessBuilder.start()* → *iniciar procesos*
  - *Runtime.exec()* → *ejecutar comandos*

RUNTIME

La clase RUNTIME

Método	Funcionalidad
Static Runtime getRuntime()	Devuelve el objeto Runtime asociado con la aplicación actual Java
Process exec(String command) Process exec(String[ ], comandoYArg) Process exec(String[ ], comandoYArg, String[ ] envp) Process exec(String[ ], cmdarray,String[ ] envp, File dir)	Ejecuta cualquier comando del SO, algunas variantes permiten especificar el entorno de ejecución y el directorio para ejecutar el comando.
void exit(int status) void halt(int status)	Termina la ejecución de la maquita virtual. El primer método lo hace de manera ordenada y el segundo de manera abrupta
int availableProcessors()	Devuelve el número de procesadores disponibles para la maquina virtual
long freeMemory()	Devuelve la cantidad de memoria disponible

Excepciones Process

Excepcion	
SecurityException	Existe un gestor de seguridad y no se permite la ejecución del subproceso
IOException	Error E/S
NullPointerException	El comando es nulo
IllegalArgumentException	El comando está vacío

Crea un programa Java que ejecute un comando de la línea de comando por ejemplo NOTEPAD

Verifica que el comando se ha ejecutado, aunque no se muestre la salida.

Concurrencia1.java

Creación de procesos con Java

- Java dispone en el paquete *java.lang* de varias clases para la gestión de procesos: **Process** y **Runtime**

- Dos métodos importantes de la clase **Runtime**:

  - ***Static Runtime getRuntime()***

    *Devuelve el objeto **Runtime** asociado a la aplicación Java en curso*

  - ***Process exec (String comando)***

    - *__Ejecuta__ la orden especificada en comando en un __proceso separado__.*
    - *Devuelve un objeto **Process** que se puede utilizar para controlar la interacción del programa Java con el nuevo proceso en ejecución.*
    - *La orden puede ser __cualquier comando del sistema operativo__.*
    - *Puede lanzar varias __excepciones__:*

      SecurityException: si existe un gestor de seguridad y no se permite la ejecución de subprocesos.

      IOException: error de E/S

      NullPointerException: el comando es nulo

      IllegalArgumentException: el comando está vacío

Uso de Runtime

- La ejecución de procesos es responsabilidad del sistema operativo, del cual nos aísla la máquina virtual Java, si bien podemos usar la clase Runtime:

1. Sólo existe una instancia de la clase Runtime, que se obtiene con:

```
Runtime rt = Runtime.getRuntime();
```

2. El objeto obtenido con la llamada a Runtime.getRuntime() nos permite ejecutar procesos con

```
rt.exec(“miprograma”);
```

En Windows, también podemos omitir el nombre del programa y

poner directamente la ruta del recurso

```
rt.exec("cmd /c start http://elvex.ugr.es/");
rt.exec("cmd /c start mailto:fberzal@decsai.ugr.es");
rt.exec("cmd /c apuntes.doc")
```

Ejemplo 1: ejecutar una aplicación de Windows

- **getRuntime()** nos devuelve el objeto **Runtime** con la aplicación Java en curso. En un proceso separado ejecuta la orden asociada a comando, abriendo el Notepad.
- **printStackTrace()** ayuda al programador a comprender dónde ocurrió el problema real.
- Es un método de la clase **Throwable** of *java.lang package*. Imprime varias líneas en la consola de salida.


```java
public class Ejemplo1 {
    public static void main(String[] args) {
        Runtime r = Runtime.getRuntime();
        String comando ="NOTEPAD";
        Process p;
        try {
            p= r.exec(comando);
        } catch (Exception e) {
            System.err.println("Error en " + comando);
            e.printStackTrace();
        }
    }


}
```

Ejecución de comandos sin ejecutable

- Para los comandos de Windows que no tienen ejecutable, por ejemplo DIR, es necesario utilizar el comando CMD.EXE

```
String comando = "CMD /C DIR";
```

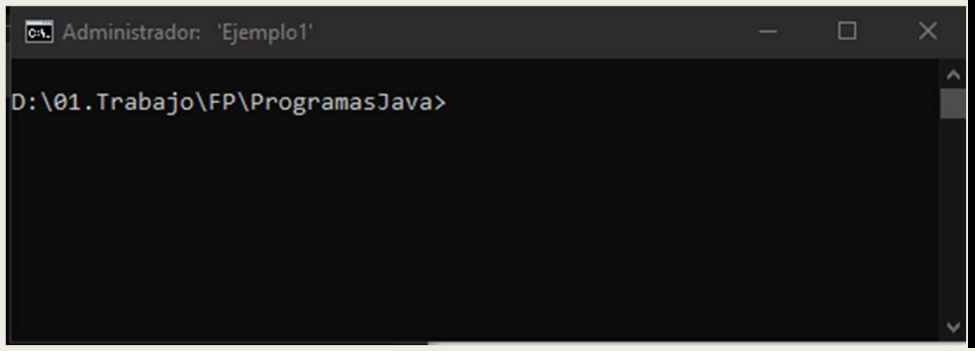
CMD /C DIR Ejecuta el comando especificado y luego finaliza

CMD /K DIR Ejecuta el comando especificado pero sigue activo

- Si en vez de NOTEPAD ponemos CMD /C DIR no obtendremos ninguna salida, porque la salida del comando se redirige a nuestro programa Java, no a la pantalla.

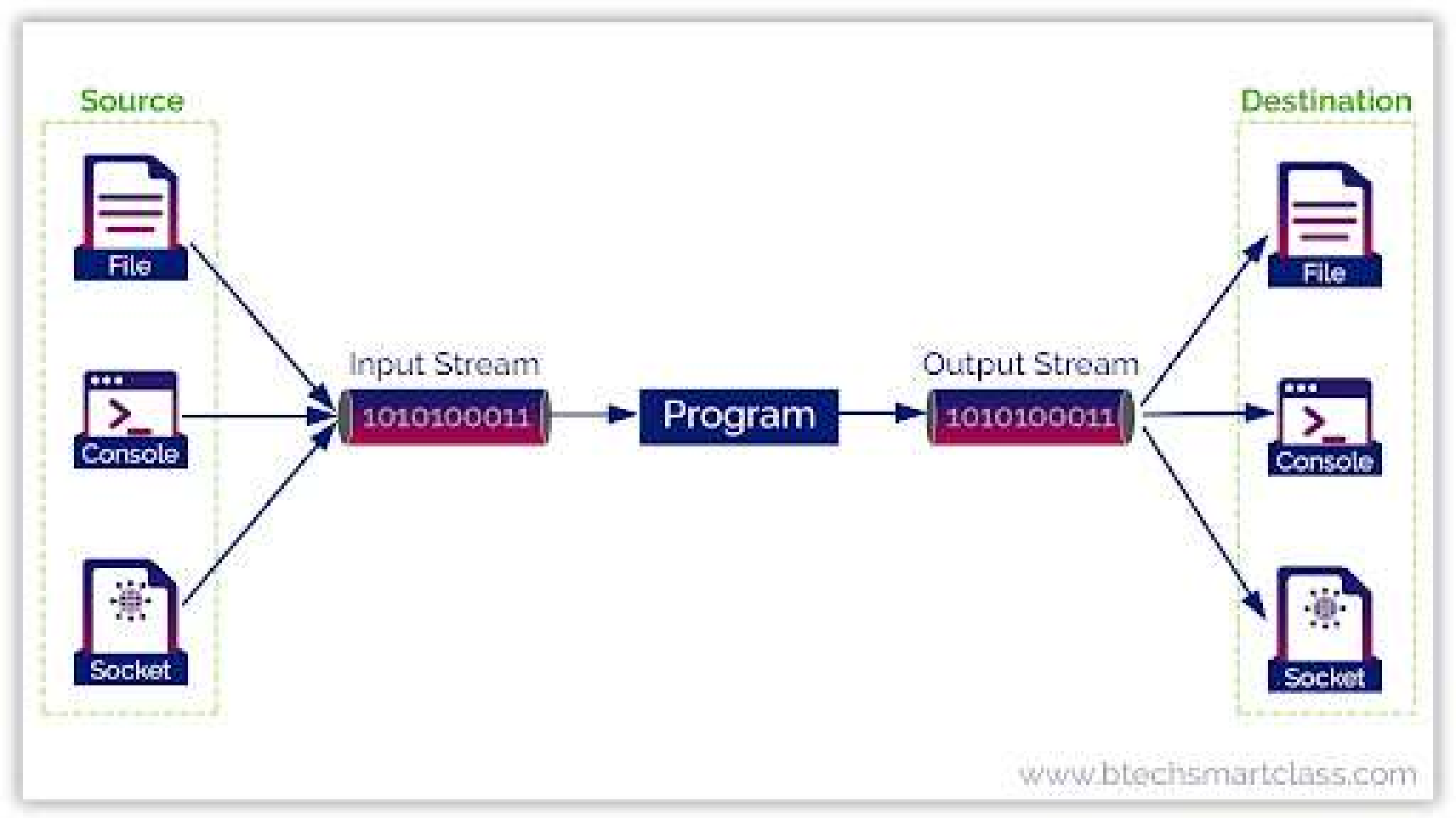
Como abrir el CMD

- El comando CMD se ejecuta en background
- Para poder abrir la ventana ejecutaremos el programa desde línea de comandos
- javac Ejemplo1.java
- \ProgramasJava>java ejemplo1/Ejemplo1




```
Runtime r= Runtime.getRuntime();
        String comando= "cmd /c start title
'Ejemplo1 Runtime CMD'";
        Process p;
        try {
            p = r.exec(comando);
        }catch (Exception e) {
            System.out.println("Error en: "
+ comando);
            e.printStackTrace();
        }
```

STREAM


Source
File
Console
Socket
Input Stream
1010100011
Program
Output Stream
1010100011
Destination
File
Console
Socket
www.btechsmartclass.com

Comunicación de procesos en java

- En Java 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 flujo padre a través de los siguientes streams:
  - *OutputStream: flujo de salida del proceso hijo.*
  - *InputStream: flujo de entrada del proceso hijo.*
  - *ErrorStream: flujo de error del proceso hijo.*

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

Comunicación entre procesos

mecanismo de comunicación entre procesos en Java cuando se usa la clase Process (o ProcessBuilder).

En pocas palabras:

- El **proceso hijo** no tiene consola propia: sus **canales de E/S** se conectan al proceso padre mediante **streams.**
- El **padre** puede **escribir en el OutputStream del hijo** (lo que equivale a la entrada estándar del hijo, stdin).
- El padre puede **leer lo que el hijo escribe en su salida (stdout)** a través de un InputStream.
- Y también puede **leer los mensajes de error (stderr)** con otro InputStream.

Entrada o lectura de datos en Java

- Lectura de datos usando las clases BufferedReader y InputStreamReader y la librería java.io

```
//Notar que readLine() nos obliga a pasar throws IOException
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
//Ya tenemos el "lector"
System.out.println("Por favor ingrese su nombre");
//Se pide un dato al usuario
String nombre = br.readLine();
//Se lee el nombre con readLine() que retorna un String con el dato
```

Clase Process

- Para leer la salida, es decir, lo que nos devuelve el método *exec()* del **Runtime**, tenemos que usar el objeto **Process,** para comunicarse con ellos usamos el método:
  - *GetInputStream() para leer la salida del proceso.*
  - *Así podremos leer lo que el comando CMD escribió en la consola.*

```
Process p = Runtime.getRuntime().exec("CMD /C DIR");
InputStream inStr = p.getInputStream();
```

  - *Para leer la salida usamos el método **Readline()** de **BufferedReader** que devuelve una línea de texto.*

Alternativa sin close() explicito

- Las clases InputStream y BufferedReader son clousable ver Javadoc
- Enlace documentación clase InputStream: https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/io/InputStream.html
- Enlace documentación clase BufferedReader
- https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/io/BufferedReader.html
- Por lo que salvo que sea necesario mantenerlo abierto como en las transacciones se puede meter la declaración dentro del try() y se auto cerrará al terminar este
- try (InputStream salidaComando = p.getInputStream()){}

Ejemplo 2: ejecución del comando DIR de Windows


```
Runtime r= Runtime.getRuntime();
        String comando= "cmd /c dir", linea= "";
        Process p= null;
        try {
            //p = r.exec(comando);
            p = r.exec(comando);
            try(InputStream salidaComando = p.getInputStream()) {
                try(BufferedReader buffer = new BufferedReader(new
InputStreamReader (salidaComando))){
                    linea = buffer.readLine();
                    while(linea != null) {
                        System.out.println(linea);
                        linea = buffer.readLine();
                    }
                }
            }
        }catch (Exception e) {
            System.err.println("Error en: " + comando);
            e.printStackTrace();
        }
```

```java
import java.io.*;

public class Ejemplo2 {
   public static void main(String[] args) {
        Runtime r= Runtime.getRuntime();
        String comando= "cmd /c dir", linea= "";
        Process p= null;

        try {
            //p = r.exec(comando);
            p = r.exec(comando);
            InputStream salidaComando = p.getInputStream();
            BufferedReader buffer = new BufferedReader(new InputStreamReader (salidaComando));
            linea = buffer.readLine();
            while(linea != null) {
                System.out.println(linea);
                linea = buffer.readLine();
            }
            buffer.close();

        }catch (Exception e) {
            System.out.println("Error en: " + comando);
            e.printStackTrace();
        }
    }
}
```

Método getErrorStream()

La clase **Process** tiene el método **getErrorStream**() nos permite obtener un Stream para leer los posibles errores que se produzcan al lanzar el proceso.

```
//Caso de error en el comando
            InputStream errorComando  = p.getErrorStream();
            buffer = new BufferedReader(new InputStreamReader (errorComando));
            linea = buffer.readLine();
            while(linea != null) {
                System.out.println(linea);
                linea = buffer.readLine();
            }
            buffer.close();
```

Espera de procesos (waitFor)

- Además de la utilización de los flujos de datos 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.
- Como resultado se recibe la información de finalización del proceso hijo, especificando dicho valor mediante un número entero que significa cómo resultó la ejecución (por convención se utiliza 0 para indicar que **el hijo** ha acabado de forma correcta).

Espera de procesos (waitFor). cont

- Mediante **waitFor()** de la clase Process el padre espera bloqueado hasta que el hijo finalice su ejecución, volviendo inmediatamente si el hijo ha finalizado con anterioridad o si alguien le interrumpe.
- Además, se puede utilizar **exitValue()** para obtener el valor de retorno que devolvió un proceso hijo.

Configurar la variable PATH para ejecutar el compilador de Java (javac) desde el intérprete de comandos

En la variable **PATH** se pueden indicar las rutas de búsqueda de archivos ejecutables en el intérprete de comandos. Por tanto, si todavía no se ha configurado la variable **PATH** y se intenta ejecutar el compilador de Java (**javac**) desde una carpeta distinta a donde está ubicado –y sin especificar dicha ubicación– en la pantalla se mostrará un mensaje haciendo saber que **javac** no se reconoce.

Configurar la variable PATH

En Windows, para que **javac.exe** sea reconocido, se pueden seguir los siguientes pasos:

- Hacer clic en el botón derecho de *"Equipo"*.
- Seleccionar *"Propiedades"*.
- Pinchar en *"Configuración avanzada del sistema"* > *"Variables de entorno"*.
- En la sección **Variables del sistema**, seleccionar la variable **PATH**.
- Pulsar en el botón *"Editar"*.
- En el *"Valor de la variable"* –sin borrar lo que ya hay– añadir al final un punto y coma "**;**" seguido de la ruta donde se encuentre el archivo **javac.exe**.

Comprobación

Abrir una nueva consola y teclear **PATH**

Actividad – redirección de salida

Crearemos un programa Java que imprima varias veces un saludo que se envía desde la línea de comandos.

El saludo se lo pasaremos por consola al ejecutarlo

Al ejecutarlo redirigiremos la salida a un fichero

ProjectoSaludo\src\saludo>javac Saludo.java

ProjectoSaludo\src>java saludo/Saludo “Hola clase”

1.Hola clase

2.Hola clase

Java Saludo “Hola clase” > fic.txt

Crearemos un programa Java que imprima varias veces un saludo que se envía desde la línea de comandos.

El saludo se lo pasaremos por consola al ejecutarlo

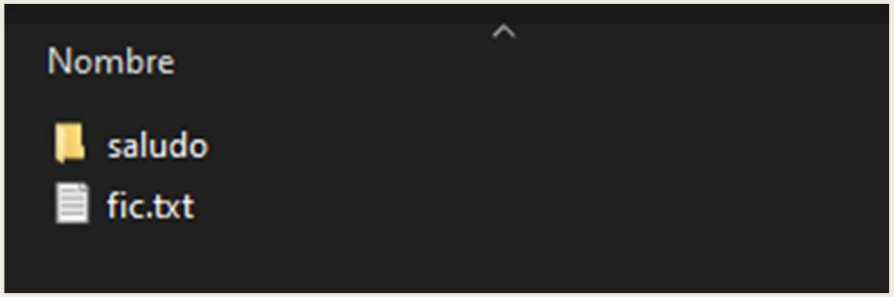
Al ejecutarlo redirigiremos la salida a un fichero

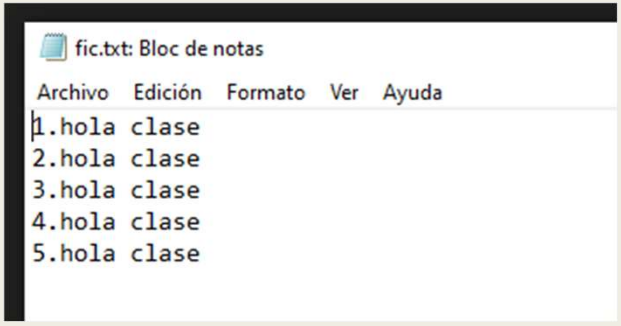
Proceso de ejecución:

Javac Saludo.java

Java Saludo "Hola clase" > fic.txt

```
NetBeansProjects\Saludo\src>java saludo/Saludo "hola clase" > fic.txt
```





EjemploSaludo

Creamos un programa Java que escribe 5 veces un saludo que se envía desde la línea de comandos:

```
public class EjemploSaludo {
    public static void main(String[] args) {
        if(args.length == 0) {
            System.out.println("Se necesita un saludo");
        }
        else {
            for (    i = 0; i <= 7; i++) {
                System.out.println(args[0]);
            }
        }
    }


}
```

Redirección de la salida a un fichero

Para redireccionar la salida de la ejecución de un comando a un fichero podemos utilizar las clases – FileOutputStream y PrintWriter

Ejecutamos el programa desde la línea de comandos

```
C:\Users\Profesor_PS\eclipse-workspace\UnSaludo\src>javac UnSaludo.java

C:\Users\Profesor_PS\eclipse-workspace\UnSaludo\src>java UnSaludo "Hola Mundo!!" > fich.txt
```

Ejecutamos el programa :
javac UnSaludo.java
java UnSaludo “Hola Mundo!!” > fich.txt
Se crea un fichero fich.txt con la salida generada por el programa.
Ejecutamos type ficht.txt para saber el contenido del fichero.

```
C:\Users\Profesor_PS\eclipse-workspace\UnSaludo\src>type fich.txt
1. Hola Mundo!!
2. Hola Mundo!!
3. Hola Mundo!!
4. Hola Mundo!!
5. Hola Mundo!!
```

Ejecutar el programa desde la línea de comandos dentro de paquete

En caso de que se definan paquetes para que te encuentre todas las clases contenidas en ese paquete se deberá compilar con –d para que te genere todas las clases en ese directorio y que luego al compilar te las encuentre donde deben estar ubicadas:

Ruta proyecto \ src\ ejercicio1> javac –d . *.java

De esta forma cuando ejecutemos el programa irá al paquete definido en el programa:

Ruta proyecto \ src\ ejercicio1> dir

```
package ejercicio1;

import java.util.Scanner;

public class Ejercicio1 {
```

Para ejecutar se pone simplemente el comando normal o con un “.” para separar el nombre del paquete:
Ruta proyecto \ src\ ejercicio1>java Padre.java 20

- Crea un programa sencillo que realice lo siguiente:
  1. *Recoja los argumentos de entrada y los muestre.*
  2. *Si no tiene argumentos de entrada debe dar un mensaje de aviso.*
- Muestra pantallazos del correcto funcionamiento del programa.

¿Por qué no se genera el fichero?

Si en el comando del ejemplo 2, le pasamos la instrucción anterior que ejecutamos en la línea de comandos:
comando = "java UnSaludo \"Hola Mundo!!\" > ficheje2.txt"
al ejecutarlo comprobamos que no se genera el fichero de salida.

Esto es porque el **método exec() no actúa como un intérprete de comandos o Shell, lo que hace es ejecutar un programa.**
Si queremos que la salida vaya a un fichero hay que añadirlo mediante programación.

Para enviar la salida a un fichero utilizamos las clases **FileOutputStream** y **PrintWriter** para enviar la información al fichero

Ejemplo3Fic

Como enviar la salida a un fichero utilizando las clases **FileOutputStream** y **PrintWriter** para enviar la información al fichero: VER Ejemplo3Fic

```
//devuelve el objeto Runtime con la aplicacion en curso
        Runtime r= Runtime.getRuntime();
        String comando= "java EjemploSaludo \"Hola Mundo!!\" ", linea = "";
        Process p= null;
        if (args.length < 1) {
        System.err.println("Se necesita un nombre de fichero");
        System.exit(1);
        }
```

Eejmplo3Fic (2)

```
try {
            //Se utilizan las siguientes clases para escribir
            //lo obtenido con el comando en el fichero
            FileOutputStream fos = new FileOutputStream(args[0]);
            try(PrintWriter pw = new PrintWriter(fos)){
            p = r.exec(comando);
            //lee el stream de salida del proceso, lo que gener en consola
                try (InputStream salidaComando = p.getInputStream()){
                    try(BufferedReader br = new BufferedReader(new
InputStreamReader (salidaComando))){
                        while((linea = br.readLine())!= null) {
                            System.out.println("Inserto en" +args[0]+" >"
+linea);
                            linea = br.readLine();
                        }
                    }
                }
            }
        }catch (Exception e) {e.printStackTrace();}
```

Ejemplo3Fic (3)

```
//conprobacion del error
        int exitVal;
        try{
            //el proceso actual espera hasta que el subproceso representado
por Process finalice
            exitVal= p.waitFor();
            System.out.println("Valor de salida" + exitVal);
        } catch (InterruptedException ie) {
            // TODO: handle exception
            ie.printStackTrace();
        }
```

Genera un programa que guarde en un fichero de nombre SVCHOST.TXT los servicios que se están ejecutando bajo el proceso svchost.exe

```
String comando = "tasklist /svc /fi \"imagename eq svchost.exe\""
```

PROCESSBUILDER

ProcessBuilder

Las versiones más recientes de Java añaden otra clase para crear y ejecutar procesos **ProcessBuilder.**

También pertenece al paquete *java.lang* como **Process** y **Runtime.**

El método **start()** crea una nueva instancia de **Process.**

```
ProcessBuilder pb = new ProcessBuilder("df –h/" ) ;
pb. start ();
```

Este código no funciona, nos da una excepción de que no existe el archivo o directorio.

Process builder **tiene constructores** para una **lista de String** o para **varios String** e interpreta **el primero** como el **nombre del programa** a ejecutar.

Ver clase ProcessBuilder.

https://docs.oracle.com/javase/8/docs/api/java/lang/ProcessBuilder.html

ProcessBuilder

Este código si funcionaria, siempre que el primero sea el programa o comando a ejecutar y el resto los argumentos.

El método **start()** crea una nueva instancia de **Process.**

```
Process p = new ProcessBuilder("Comando", "Arg1").start();
```

Ejemplo:

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

O bien:

```
ProcessBuilder pb = new ProcessBuilder("CMD", "/C", "DIR").start();
ProcessBuilder pb = new ProcessBuilder("java", "UnSaludo", "\"Hola Mundo!!\"").start();
```

ProcessBuilder

Cada **ProcessBuilder** gestiona los siguientes atributos de un proceso:

- *comando:*
  - Un comando: es una lista de Strings que representan el ejecutable invocado y sus argumentos.
- *entorno (environment) con sus variables.*
- *directorio de trabajo.*
  - *Un directorio de trabajo. El valor por defecto es el directorio de trabajo del proceso en curso.*
- *fuente de entrada estándar.*
  - *El subproceso hijo, por defecto, lee la entrada de una tubería, puede acceder a ella a través de **Process.getInputStream()**. La entrada estándar puede redirigirse a otra fuente con **redirectInput**()*
- *Destino para la salida estándar y salida de error.*
  - *Se puede acceder a ellas con **Process.getOutputStream()** y **Process.getErrorStream()**. Pueden redirigirse a otros destinos con **redirectOutput**() y **redirectError**() que nos permiten redirigir la salida estándar y de error a otro fichero.*

ProcessBuilder: métodos **environment**() y **command**()

```
public class PBEntorno {
    public static void main(String[] args) {
        ProcessBuilder pb = new ProcessBuilder();
        //El metodo enviroment devuelve las variables de entorno del
proceso
        // los objetos Map asocian claves a valores
        Map entorno = pb.environment();
        System.out.println(" VAriables del entorno");
        System.out.println(entorno);
        pb = new ProcessBuilder("java","UnSaludo", "\"Hola Mundo!!\"");
        // el metodo command devuelve el nombre del proceso y sus
argumentos
        List<String> lista = pb.command();
        System.out.println("Argumentos del comando");
        for (String cadena : lista)
            System.out.println(cadena);
    }
}
```

ProcessBuilder: ejecutar el comando DIR con **command**

```
ProcessBuilder pb = new ProcessBuilder();
        pb = pb.command("CMD","/C", "DIR");
        try{
            Process p = pb.start();
            //La calse InputStream representa un flujo ordenado de bytes
            InputStream input = p.getInputStream();
            try(InputStreamReader ir = new InputStreamReader(input)){
                try(BufferedReader buffer = new BufferedReader(ir)){
                    String linea;
                    while ((linea = buffer.readLine())!=null){
                        System.out.println(linea);
                    }
                }
            }
        }
        catch (Exception e) {
            // TODO: handle exception
            e.printStackTrace();
```

```java
public class PBRedirect {
    public static void main(String[] args) {
        ProcessBuilder pb = new ProcessBuilder();
        pb = pb.command("CMD","/C", "DIR");
        try{
            File fOut = new File ("C:/Users/Profesor_PS/Procesos/salida.txt");
            File fErr = new File ("C:/Users/Profesor_PS/Procesos/error.txt");
            pb.redirectOutput(fOut);
            pb.redirectError(fErr);
            pb.start();
        }
        catch (Exception e) {
            // TODO: handle exception
            e.printStackTrace();
        }
    }
}
```

ProcessBuilder: métodos **redirectInput**()

También podemos ejecutar varios comandos del sistema operativo que estén dentro de un fichero bat. Se utiliza el método **redirectInput**() para indicar que la entrada al proceso se encuentra en un fichero.

Crea un archivo .bat

- Crea un archivo .bat con el nombre fichero.bat
- El archivo deberá realizar las siguientes operaciones
  - *Crear una carpeta*
  - *Crear un fichero dentro de la carpeta con el texto que queráis*
  - *Mostrar el directorio*
- Este archivo se utilizará como entrada en el siguiente programa

Ejemplo redirect entrada BAT

```
public class PBRedirectBat {
    public static void main(String[] args) {
        ProcessBuilder pb = new ProcessBuilder();
        pb = pb.command("CMD");
        try{
            File fOut = new File ("C:/Users/Profesor_PS/Procesos/salida.txt");
            File fErr = new File ("C:/Users/Profesor_PS/Procesos/error.txt");
            File fIn = new File("C:/Users/Profesor_PS/Procesos/fichero.bat");

            pb.redirectInput(fIn);
            pb.redirectOutput(fOut);
            pb.redirectError(fErr);
            pb.start();
        }
        catch (Exception e) {
            // TODO: handle exception
            e.printStackTrace();
        }
    }
}
```

Lectura ProcessBuilder

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

        BufferedReader buffer= null;
        String linea;

        InputStreamReader input= new InputStreamReader(System.in);
        //Como las variables scanner
        try {
            buffer = new BufferedReader(input);
            System.out.println("Introduce una cadena de texto: ");
            linea = buffer.readLine();
            System.out.println("La linea es: "+linea);
            buffer.close();
        }catch (Exception e) {
            System.out.println("Error en: " );
            e.printStackTrace();

        }
    }
}
```

ProcessBuilder: comunicación entre procesos

```java
public class PBComunicaProcess{
    public static void main(String[] args) {
        ProcessBuilder pb = new ProcessBuilder("java", "LecturaPB.java");
        //redireccionamos la salida de error
        pb.redirectErrorStream();
        try {
            Process p= pb.start();
            //abrimos la comunicacion para escribir en el Stream
            OutputStream o = p.getOutputStream();
            //en el stream solo se pueden introducir bytes o int
            o.write("Hola Clase \n".getBytes());
            //liberamos el stream de escritura
            o.flush();
            int revisoError = p.waitFor();
            //al terminar el proceso verificamos la salida
            System.out.println("Erro igual a: " + revisoError);
        } catch (Exception e) {
            System.err.println("Error en: ");
            e.printStackTrace();
        }
    }
}
```

La Clase Process

- La clase Process tiene métodos que devuelven streams asociados a la entrada estándar del proceso, a la salida estándar y a la salida de error.
- Esto permite enviar datos a la entrada estándar del proceso y leer la salida estándar y de error, lo que permite el control total del proceso.

Metodos de la Clase Process

Método	Funcionalidad
InputStream getImputStream()	Devuelve un stream de entrada conectado con la salida estándar del proceso
OutputStream getOutputStream()	Devuelve un stream de salida conectado con la entrada estándar del proceso
InputStream() getErrorStream()	Devuelve un stream de entrada conectado con la salida de error del proceso

Ejemplo de uso

- Creamos un programa que utilizando la clase ProcessBuilder ejecute un comando que se le pasara por argumento al programa.
- La salida se recogerá con getInputStream
- Se contruira un Buffered Reader para poder obtener la salida y leerla línea a línea.