



MÓDULO PSP

UDI: PROGRAMACIÓN CON PROCESOS

DAM – 2º curso

PSP

Contenido

- Conceptos básicos:
 - Programa
 - Proceso
- Comandos en Linux para la gestión de procesos
- Conceptos básicos:
 - Programación concurrente
 - Programación paralela
 - Programación distribuida.
- Clase Runtime
- Clase Process
- Clase ProcessBuilder

PROCESOS Y PROGRAMACIÓN CONCURRENTE



CONCEPTOS BÁSICOS

Una cocinera que tenga intención de hacer un plato elaborado, con salsas y guarniciones diferentes, probablemente utilizará diferentes sartenes para cocinar cada elemento que necesite utilizar para elaborar el plato.



Tendrá una sartén con la salsa, otra preparando la guarnición y en otra cocinará el plato principal. Todo esto lo hará al mismo tiempo y, finalmente, cuando termine todas estas tareas las juntará en un único plato.

CONCEPTOS BÁSICOS

Esta analogía nos sirve para introducir el término **multiproceso**.

La cocinera está ejecutando diferentes procesos a la vez: cocina la salsa, la guarnición y el plato principal, siguiendo un orden de ejecución para llegar a un resultado que esperaba, un buen plato.

CONCEPTOS BÁSICOS

Siguiendo con la analogía de la cocinera, un **proceso** sería la actividad realizada por la cocinera (**procesador**) de elaborar alguno de los complementos o el plato principal (**resultados**) a partir de una receta (**programa**).

Cada proceso de creación requiere coger los ingredientes (**los datos**), ponerlos en la sartén (**recurso asociado**) y cocinarlos con independencia del resto de procesos.

PROGRAMA

¿Qué es un programa?

PROGRAMA

Un programa es un conjunto de instrucciones que describen el tratamiento a dar a unos datos iniciales (de entrada) para conseguir el resultado esperado (una salida concreta) .

Es, por tanto, un elemento estático o pasivo.

PROCESO

¿Qué es un **proceso**?

PROCESO

Definición informal:

Un **proceso** es un programa en ejecución.

Es, por tanto, un elemento **dinámico** o activo.

PROCESO

El **sistema operativo** es el encargado de la gestión de procesos. Los crea, los elimina y los provee de instrumentos que permiten la ejecución y también la comunicación entre ellos.

Cuando se ejecuta un programa, el sistema operativo crea una instancia del programa: **el proceso**.

Si el programa se volviera a ejecutar se crearía una nueva instancia totalmente independiente a la anterior (un nuevo proceso).

PROCESO

Un **proceso es un programa en ejecución**. Por ejemplo, podemos escribir una aplicación Java y guardarla en el disco. Esto se considera un programa y una entidad pasiva.

Sin embargo, cuando ejecutamos el programa, el sistema operativo crea un proceso Java. Un proceso es una entidad activa mientras está en ejecución.

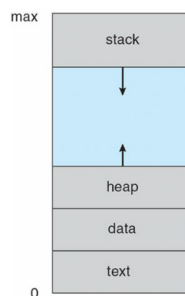
PROCESO: MEMORIA ASIGNADA

Cuando un programa se está ejecutando, ocupa memoria. A veces ni siquiera somos conscientes de la memoria que se está asignando. De hecho, cada vez que crea una nueva variable, su programa le asigna más memoria para almacenar esa variable

Una de las cosas de las que es responsable la **JVM** es la **gestión de la memoria**. Digamos que creas una variable o un objeto en una aplicación Java. ¿Dónde se almacena? ¿Cuánto tiempo 'dura'? ¿Cuándo se elimina de la memoria?

PROCESO: MEMORIA ASIGNADA

El proceso se carga en la memoria principal cuando se ejecuta el programa. En la memoria principal, un proceso tiene una **pila**, un **montículo**, **datos** y **texto**.

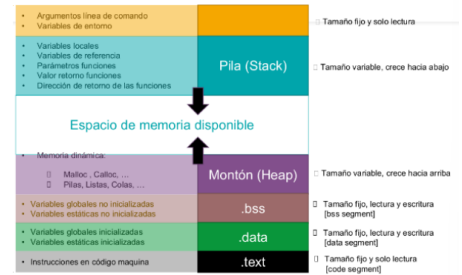


PROCESO: MEMORIA ASIGNADA



¿Cómo se ve un proceso en memoria? Los segmentos de memoria son:

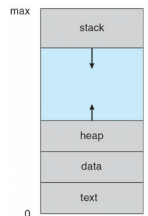
- **Texto:** Almacena el código siendo ejecutado. La imagen binaria del programa.
- **Datos:** almacena variables globales, separadas en inicializadas y no inicializadas
- **Montículo (heap):** memoria de trabajo. Cuando un programa solicita memoria dinámica para una variable se le asigna memoria de este segmento.
- **Pila (stack):** se almacenan las variables locales. Se utiliza para preservar el estado en la invocación anidada de funciones.



En detalle

Cada segmento tiene asignado una "cosa" del proceso.

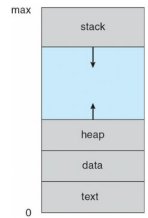
PROCESO: MEMORIA ASIGNADA



Para identificar cada ubicación de memoria en la memoria de un programa, asignamos a cada byte de memoria una "dirección". Las direcciones van desde 0 hasta la dirección más grande posible, dependiendo de la máquina. Como se muestra en la siguiente figura, los segmentos **texto**, **datos** y **heap** tienen números de direcciones bajos, mientras que la **pila** direcciones más altas.

Por convención, expresamos estas direcciones en números de base 16. Por ejemplo, la dirección más pequeña posible 0x00000000 (donde 0x significa base 16) y la dirección más grande posible podría ser 0xFFFFFFFF.

PROCESO EN MEMORIA: PILA

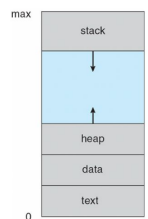


Como se muestra arriba, el segmento de la pila está cerca de la parte superior de la memoria con una dirección alta.

Cada vez que se llama a una **función**, la máquina le asigna algo de memoria de pila. Cuando se declara una nueva variable local, se asigna más memoria de pila para que esa función almacene la variable. Estas asignaciones hacen que la pila crezca hacia abajo. Después de que la función regresa, la memoria de pila de esta función se desasigna, lo que significa que todas las variables locales dejan de ser válidas.

La asignación y desasignación de la memoria de la pila se realiza automáticamente. Las variables asignadas en la pila se denominan variables de pila o variables automáticas.

PROCESO EN MEMORIA: PILA



Se llama **pila** de ejecución porque es una estructura de datos que funciona bajo el concepto de **LIFO** (*last in first out*).

Otras características:

- Crece y se reduce a medida que se llaman y devuelven nuevos métodos, respectivamente.
- Las variables dentro de la pila existen sólo mientras se esté ejecutando el método que las creó.
- Se asigna y desasigna automáticamente cuando el método finaliza la ejecución.
- Si esta memoria está llena, Java arroja `java.lang.StackOverflowError`.
- El acceso a esta memoria es rápido en comparación con la memoria del montículo.
- Esta memoria es **segura** para subprocesos, ya que cada **subproceso** opera en su propia pila.

PROCESO EN MEMORIA: HEAP

El montículo libre, zona libre, almacenamiento libre o heap es una estructura dinámica de datos utilizada para almacenar datos en ejecución.

Este espacio de almacenamiento dinámico se utiliza para la asignación de memoria dinámica de **objetos Java** y clases JRE en tiempo de ejecución. Los nuevos objetos siempre se crean en el espacio del montículo y las referencias a estos objetos se almacenan en la memoria de la pila.

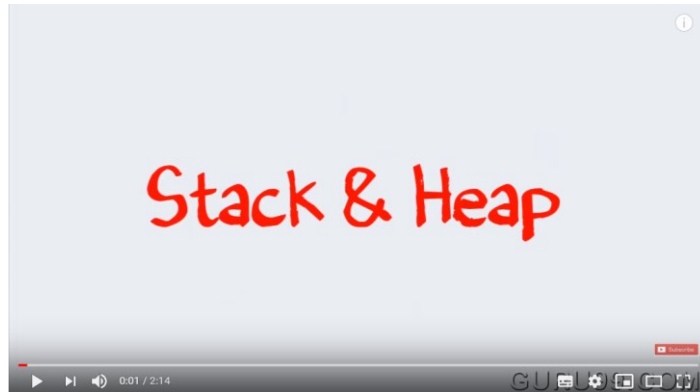
PROCESO EN MEMORIA: HEAP

Otras características:

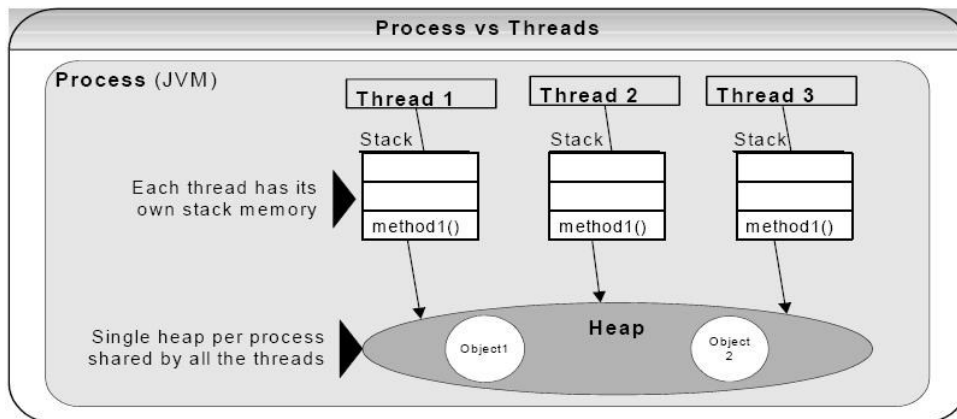
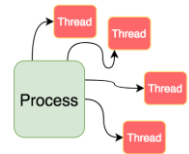
- Si el espacio del montón está lleno, Java arroja `java.lang.OutOfMemoryError`.
- El acceso a esta memoria es comparativamente más lento que la memoria de pila.
- Esta memoria, a diferencia de la pila, no se desasigna automáticamente. Necesita Garbage Collector para liberar objetos no utilizados a fin de mantener la eficiencia del uso de la memoria.
- A diferencia de la pila, un montículo **no es seguro** para **subprocesos** y debe protegerse sincronizando adecuadamente el código.

JAVA STACK AND HEAP - JAVA MEMORY MANAGEMENT

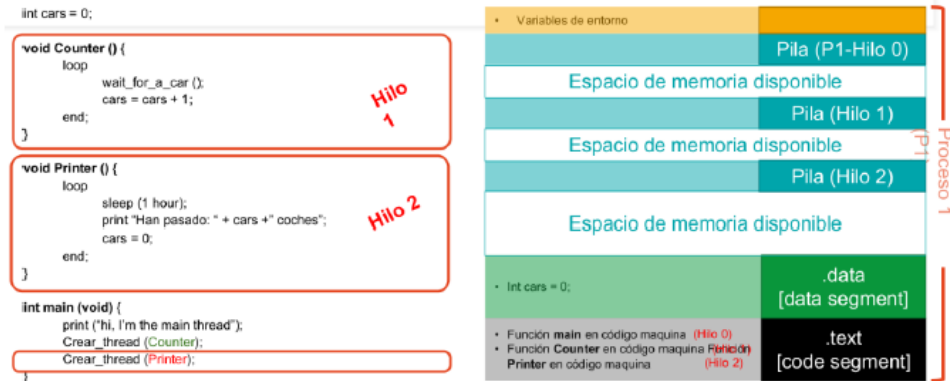
https://www.youtube.com/watch?time_continue=4&v=450maTzSlvA



CUANDO VEAMOS HILOS...



CUANDO VEAMOS HILOS...



BLOQUE DE CONTROL DE PROCESO

Cuando el proceso es creado por el **sistema operativo**, crea una estructura de datos para almacenar la información de ese proceso. Esto se conoce como **Bloque de control de proceso** (BCP). Estos bloques se almacenan en una memoria especialmente reservada para el sistema operativo conocida como espacio del kernel.

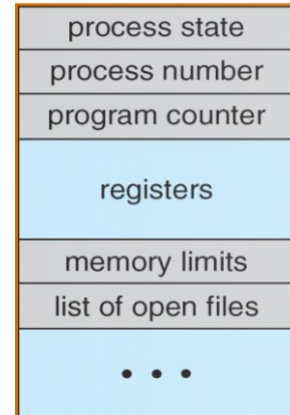
BCP es único para cada proceso y consta de varios atributos:

- Identificador del proceso (**PID**, process identifier)
- Estado** del proceso: nuevo, listo, en ejecución, en espera o finalizado.
- Prioridad**: valor numérico. Cuanto menor sea el valor, mayor será la prioridad de ese proceso.
- Contador del programa (program counter, o **PC**) es un registro del procesador que contiene la dirección la siguiente instrucción a ejecutar.
- Registros de la CPU**: acumulador, base y registros de propósito general.

process state
process number
program counter
registers
memory limits
list of open files
...

BLOQUE DE CONTROL DE PROCESO

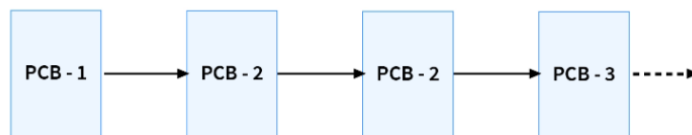
- Límites de **memoria**: este campo contiene información sobre el sistema de administración de memoria utilizado por el sistema operativo. Esto puede incluir tablas de páginas, tablas de segmentos, etc.
- Lista de **archivos abiertos**: esta información incluye la lista de archivos abiertos para un proceso
- Otra información adicional.



BLOQUE DE CONTROL DE PROCESO

¿Cómo se almacenan los BCP?

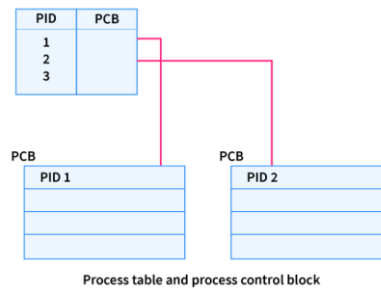
Los BCPs se almacenan en forma de LinkedList en la memoria como se muestra en la figura.



Organization of PCBs in memory

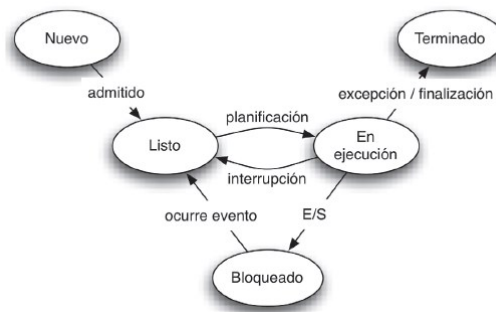
BLOQUE DE CONTROL DE PROCESO

El Sistema Operativo utiliza la [Tabla de Procesos](#) para encontrar el BCP presente en memoria. La tabla de Procesos es una tabla que contiene el ID del Proceso y la referencia al BCP correspondiente en memoria. Podemos visualizar la tabla de Procesos como un diccionario que contiene la lista de todos los procesos en ejecución.

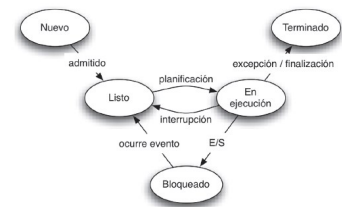


ESTADOS DE UN PROCESO

Un proceso, desde su creación hasta su finalización, pasa por diferentes estados. Generalmente, un proceso puede estar presente en uno de los [5 estados](#) durante su ejecución:

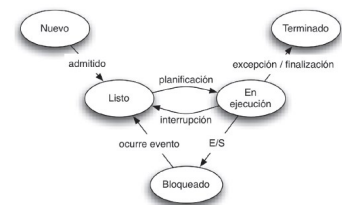


ESTADOS DE UN PROCESO



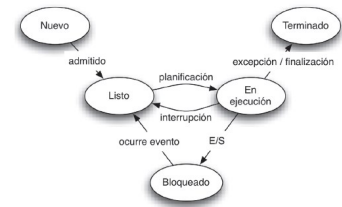
- **Nuevo:** el proceso se acaba de crear, pero aún no ha sido admitido en el grupo de procesos ejecutables por el sistema operativo. En el momento que el sistema operativo considere que hay suficientes recursos para ejecutar ese proceso pasará a listo.
- **Listo:** el proceso está listo para ser ejecutado y que se encuentra actualmente en la memoria principal del sistema. 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 se ejecutará, por lo que es el que indica cuando el proceso pasa a ejecución.

ESTADOS DE UN PROCESO



- **En ejecución:** el proceso está siendo ejecutado por la CPU en nuestro sistema. 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, pasará a estado bloqueado. 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, el sistema detiene el proceso y lo pasa al estado Listo, seleccionando otro proceso para que continúe su ejecución.

ESTADOS DE UN PROCESO



- **Bloqueo o espera:** 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 para al estado Listo y tiene que ser planificado de nuevo por el sistema.
- **Finalizado:** 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).

COMANDOS LINUX PARA LA GESTIÓN DE PROCESOS

Mediante el comando `ps` (process status) podemos ver parte de la información asociada a cada proceso.

El siguiente ejemplo muestra los procesos actualmente vivos en la máquina. Dos procesos ejecutándose, uno es el Shell y otro es la ejecución del comando `ps`.

```

gaby@ies-VirtualBox:~/Documentos/src$ ps
  PID TTY          TIME CMD
 1458 pts/1        00:00:00 ps
 3508 pts/1        00:00:00 bash

```


COMANDOS LINUX PARA LA GESTIÓN DE PROCESOS

```
gaby@ies-VirtualBox:~/Documentos/src$ ps
  PID TTY          TIME CMD
 1458 pts/1        00:00:00 ps
 3508 pts/1        00:00:00 bash
```

PID: identificador del proceso

TTY: terminal asociado del que lee y al que escribe

TIME: tiempo de ejecución asociado, es la cantidad total de CPU que el proceso ha utilizado desde que nació

CMD: nombre del proceso

COMANDOS LINUX PARA LA GESTIÓN DE PROCESOS

El comando `ps -f` muestra más información

```
gaby@ies-VirtualBox:~/Documentos/src$ ps -f
UID          PID  PPID  C  STIME TTY          TIME CMD
gaby        1464   3508  0  15:11 pts/1        00:00:00 ps -f
gaby        3508   3498  0  12:22 pts/1        00:00:00 bash
```

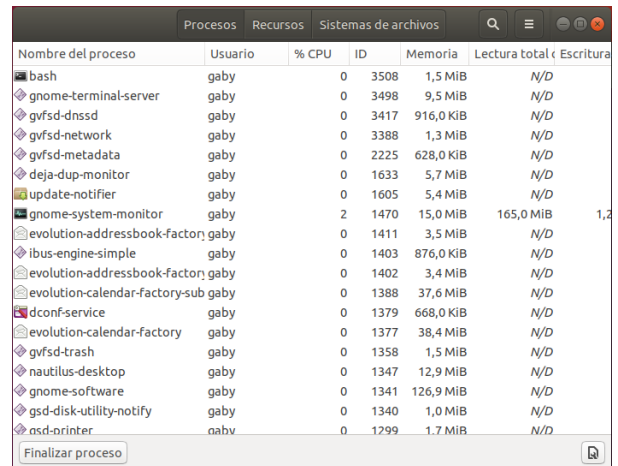
UID: nombre del usuario

PPID: el identificador del padre de cada proceso

STIME: hora de inicio del proceso

COMANDOS LINUX PARA LA GESTIÓN DE PROCESOS

En **Ubuntu** escribiendo desde la terminal **gnome-system-monitor** podemos acceder a la interfaz gráfica que nos muestra información sobre los procesos que se están ejecutando.

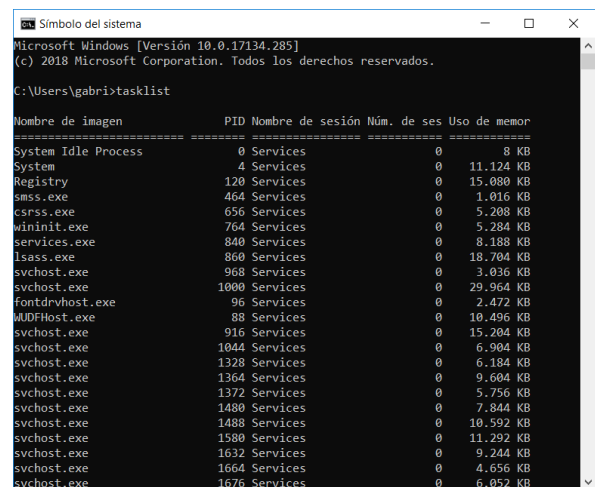


Nombre del proceso	Usuario	% CPU	ID	Memoria	Lectura total (Escritura
bash	gaby	0	3508	1,5 MiB	N/D	
gnome-terminal-server	gaby	0	3498	9,5 MiB	N/D	
gvfsd-dnssd	gaby	0	3417	916,0 KiB	N/D	
gvfsd-network	gaby	0	3388	1,3 MiB	N/D	
gvfsd-metadata	gaby	0	2225	628,0 KiB	N/D	
deja-dup-monitor	gaby	0	1633	5,7 MiB	N/D	
update-notifier	gaby	0	1605	5,4 MiB	N/D	
gnome-system-monitor	gaby	2	1470	15,0 MiB	165,0 MiB	1,2
evolution-addressbook-factory	gaby	0	1411	3,5 MiB	N/D	
ibus-engine-simple	gaby	0	1403	876,0 KiB	N/D	
evolution-addressbook-factory	gaby	0	1402	3,4 MiB	N/D	
evolution-calendar-factory-sub	gaby	0	1388	37,6 MiB	N/D	
dconf-service	gaby	0	1379	668,0 KiB	N/D	
evolution-calendar-factory	gaby	0	1377	38,4 MiB	N/D	
gvfsd-trash	gaby	0	1358	1,5 MiB	N/D	
nautilus-desktop	gaby	0	1347	12,9 MiB	N/D	
gnome-software	gaby	0	1341	126,9 MiB	N/D	
gsd-disk-utility-notify	gaby	0	1340	1,0 MiB	N/D	
nsd-nrprinter	naabv	0	1299	1,7 MiB	N/D	

COMANDOS LINUX PARA LA GESTIÓN DE PROCESOS

En **Windows** podemos usar desde la línea de comandos la orden **tasklist** para ver los procesos que se están ejecutando

Aunque lo más típico es utilizar la combinación de teclas **CTRL+Alt+Supr** para que se muestre la pantalla que da acceso al **Administrador de tareas**



```

Microsoft Windows [Versión 10.0.17134.285]
(c) 2018 Microsoft Corporation. Todos los derechos reservados.

C:\Users\gabri>tasklist

Nombre de imagen                PID Nombre de sesión Núm. de ses Uso de memor
=====
System Idle Process              0 Services              0           8 KB
System                           4 Services              0      11.124 KB
Registry                        120 Services              0     15.080 KB
smss.exe                        464 Services              0       1.016 KB
csrss.exe                       656 Services              0      5.208 KB
wininit.exe                     764 Services              0      5.284 KB
services.exe                   840 Services              0      8.188 KB
lsass.exe                       860 Services              0     18.704 KB
svchost.exe                     968 Services              0       3.036 KB
svchost.exe                    1000 Services              0     29.964 KB
fontdrvhost.exe                 96 Services              0       2.472 KB
WUDFHost.exe                   88 Services              0      10.496 KB
svchost.exe                     916 Services              0     15.204 KB
svchost.exe                    1044 Services              0      6.904 KB
svchost.exe                     1228 Services              0      6.184 KB
svchost.exe                     1364 Services              0      9.604 KB
svchost.exe                     1372 Services              0      5.756 KB
svchost.exe                     1480 Services              0      7.844 KB
svchost.exe                     1488 Services              0     10.592 KB
svchost.exe                     1580 Services              0     11.292 KB
svchost.exe                     1632 Services              0      9.244 KB
svchost.exe                     1664 Services              0      4.656 KB
svchost.exe                     1676 Services              0     6.052 KB

```

Administrador de tareas				
Archivo Opciones Vista				
Procesos Rendimiento Historial de aplicaciones Inicio Usuarios Detalles Servicios				
Nombre	Estado	3% CPU	85% Memoria	1% Disco
> Administrador de tareas		0,7%	26,2 MB	0,1 MB/s
> VirtualBox Manager		0,7%	87,4 MB	1,7 MB/s
HPNetworkCommunicatorCom		0,5%	2,4 MB	0 MB/s
> Google Chrome (22)		0,2%	461,8 MB	0 MB/s
WD App Manager		0,2%	19,7 MB	0 MB/s
> WD Drive Service (32 bits)		0,2%	5,7 MB	0 MB/s
> Explorador de Windows		0,1%	28,7 MB	0 MB/s
Application Frame Host		0,1%	5,5 MB	0 MB/s
> SmartByte Network Service		0,1%	19,1 MB	0 MB/s
> Host de servicio: Llamada a procedimiento rem...		0,1%	6,7 MB	0 MB/s
> Host del servicio: Detección SSDP		0,1%	0,8 MB	0 MB/s
System		0%	0,1 MB	1,9 MB/s
> Antimalware Service Executable		0%	101,5 MB	0,1 MB/s
> Microsoft Outlook Communications		0%	0,1 MB	0,1 MB/s

Menos detalles Finalizar tarea

ACTIVIDAD. COMANDOS

1. Comandos Linux para la gestión de procesos

PROCESADOR

Hasta ahora hemos hablado de elementos de software, tales como programas y procesos, pero no del hardware utilizado para ejecutarlos: el **procesador**.

Un **procesador** es el componente de hardware de un sistema informático encargado de ejecutar las instrucciones y procesar los datos.

PROCESADOR

¿Qué es un sistema **monoprocesador** y un sistema **mutiprocador**?

PROCESADOR

Un **sistema monoprocesador** es aquel que está formado únicamente por un procesador.

Un **sistema multiprocesador** está formado por más de un procesador.

CONCURRENCIA

¿Qué entiendes por **conurrencia**?

PROCESOS CONCURRENTES

El diccionario de la Real Academia Española nos muestra varias acepciones de la palabra **concurrentia**

Nos quedamos con la tercera:

3. *Acaecimiento o coincidencia de varios sucesos o cosas a un mismo tiempo.*

Si sustituimos sucesos por procesos tenemos una aproximación de lo que es la **concurrentia** en informática.

PROCESOS CONCURRENTES

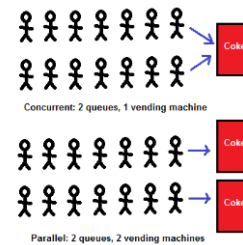
Dos **procesos** serán **concurrentes** cuando existe solapamiento o intercalado en la ejecución de sus instrucciones en un período de tiempo.

Los procesos concurrentes pueden darse en:

- Sistemas con una CPU (Ejecución **pseudo-paralela**).
- Sistemas con varias CPUs (Ejecución **paralela**).

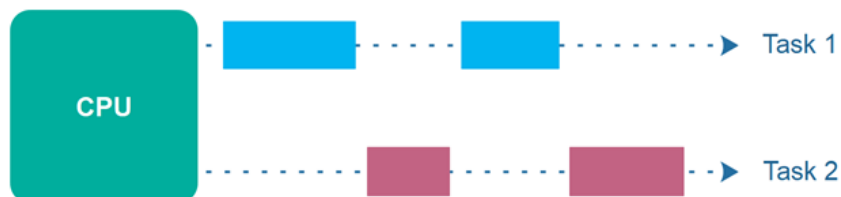
PROCESOS CONCURRENTES

Actualmente, la mayoría de los sistemas operativos aprovechan los tiempos de reposo de los procesos, cuando esperan por ejemplo, algún dato del usuario o se encuentran pendientes de alguna operación de E/S, para introducir en el procesador otro proceso, **simulando así una ejecución paralela.**



PROCESOS CONCURRENTES

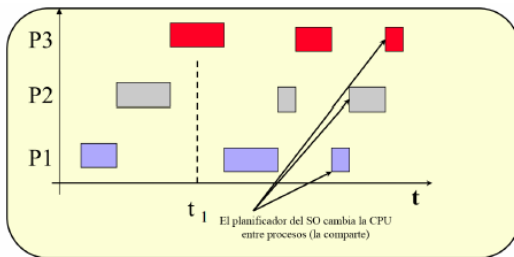
Ejecución de procesos concurrentes



Ejecución concurrente

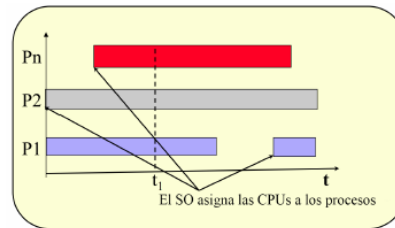
p_1 y p_2 son procesos concurrentes si la 1ª instrucción de p_2 se ejecuta entre la 1ª y la última instrucción de p_1 .

Pseudoparalelismo



El **pseudoparalelismo** son varios procesos compartiendo una CPU.

Paralelismo



El **paralelismo** son varios procesos corriendo en varias CPUs.

PROCESOS CONCURRENTES

Hablamos de **multiprogramación** cuando el sistema operativo gestiona la ejecución de **procesos concurrentes** a un sistema **monoprocesador**.

Una **aplicación concurrente** implementa procesos concurrentes. Suelen más eficientes que una aplicación que no implementa concurrencia.

PROCESOS CONCURRENTES

Cuando varios procesos se ejecutan concurrentemente puede haber varios procesos que colaboren para un determinado fin, y otros que compitan por recursos del sistema.

Estas tareas de colaboración y competencia por los recursos exigen mecanismos de **comunicación y sincronización** entre procesos.

PROGRAMACIÓN CONCURRENTES, PARALELA Y DISTRIBUIDA

En un sistema informático **multiprocesador** existen dos o más procesadores, por lo tanto se pueden **ejecutar simultáneamente varios procesos**. Existe **paralelismo real entre los procesos**.

PROGRAMACIÓN CONCURRENTE, PARALELA Y DISTRIBUIDA

Antes mencionamos que si la programación concurrente se da en un computador con un único procesador hablaremos de **multiprogramación**.

En cambio, si el computador tiene más de un procesador y, por tanto, los procesos se pueden ejecutar de forma realmente simultánea, hablaremos de **programación paralela**.

PROGRAMACIÓN CONCURRENTE, PARALELA Y DISTRIBUIDA

Es habitual que el número de procesos que se esté ejecutando de forma concurrente sea **mayor** que el número de procesadores. Por lo tanto será obligatorio que algunos procesos se ejecuten sobre el mismo procesador.

PROGRAMACIÓN CONCURRENTE, PARALELA Y DISTRIBUIDA

Un tipo especial de programación paralela es la llamada **programación distribuida**. Esta programación se da en sistemas informáticos distribuidos.

Un sistema distribuido está formado por un **conjunto de ordenadores** que pueden estar situados **en lugares geográficos** diferentes unidos entre ellos a través de una **red de comunicaciones**.

PROGRAMACIÓN CONCURRENTE, PARALELA Y DISTRIBUIDA

La **programación distribuida** es un tipo de programación concurrente en la que los procesos son ejecutados en una red de procesadores autónomos o en un sistema distribuido. Es un sistema de computadores independientes que desde el punto de vista del usuario del sistema se ve como una sola computadora.

Clases Java para la gestión de subprocesos

ACTIVIDAD. JAVA DESDE LA LÍNEA DE COMANDOS

2. Ejecutar una aplicación java desde la línea de comandos.

CLASE RUNTIME

API - Runtime class Java 11

Cada aplicación Java tiene una única instancia de clase Runtime que permite que la aplicación interactúe con el entorno en el que se ejecuta. Una aplicación no puede crear su propia instancia de esta clase.

CLASE RUNTIME

Tiene varios métodos pero los más utilizados son:

- `getRuntime()`

Devuelve el objeto de tiempo de ejecución asociado con la aplicación Java actual.

- `exec(String command)`

Ejecuta el commando especificado por la cadena en un proceso separado. De este método existe una sobrecarga, ya que existen varias versiones del mismo método pero con diferentes parámetros.

```
//Obteniendo el objeto Runtime y luego ejecutando
//el programa notepad
Runtime rt = Runtime.getRuntime();
rt.exec("notepad");

//En una línea
Runtime.getRuntime().exec("notepad");

//Obteniendo una referencia al objeto Process creado por
//el método exec
Process p = Runtime.getRuntime().exec("notepad");
```

CLASE RUNTIME

▪ gc()

Ejecuta el recolector de basura. La llamada a este método sugiere que la máquina virtual Java se esfuerce en reciclar los objetos no utilizados para que la memoria que ocupan esté disponible para su rápida reutilización. Cuando el control vuelve de la llamada al método, la máquina virtual ha hecho su mejor esfuerzo para reciclar todos los objetos descartados.

▪ availableProcessors()

Devuelve el número de procesadores disponibles para la máquina virtual Java. Este valor puede cambiar durante una invocación concreta de la máquina virtual. Por lo tanto, las aplicaciones sensibles al número de procesadores disponibles deberían sondear ocasionalmente esta propiedad y ajustar su uso de recursos adecuadamente.

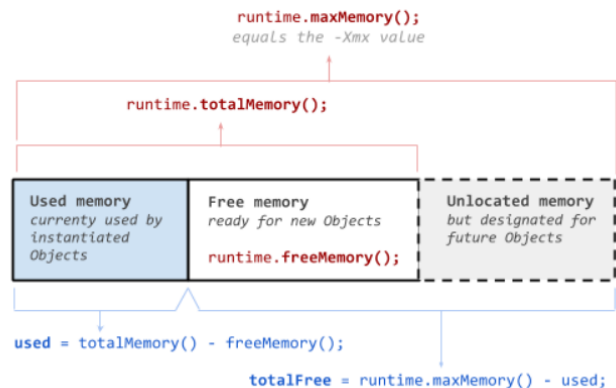
CLASE RUNTIME

Estadísticas de memoria de la máquina virtual Java

maxMemory: cantidad de memoria máxima que la JVM se ha configurado que puede utilizar

totalMemory: memoria total que la JVM tiene asignada hasta el momento

freeMemory: memoria libre del total asignado



ACTIVIDAD CLASE RUNTIME

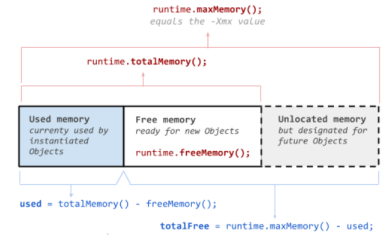
3. Copiar y ejecutar [RuntimeDemo](#) y [MemoryTest](#)

ACTIVIDAD CLASE RUNTIME

```
import java.io.IOException;

class RuntimeDemo {
    public static void main(String args[]) {
        Runtime rt = Runtime.getRuntime();
        Process proc;
        String cmd = "notepad";
        try {
            proc = rt.exec(cmd);
            proc.waitFor(); //probar ejecutar sin esta línea
        } catch (IOException ex1) {
            System.out.println(cmd + " es un comando desconocido");
        } catch (InterruptedException ex2) {
            System.out.println("Ups");
        }
    }
}
```

ACTIVIDAD CLASE RUNTIME



```
public class MemoryTest{
    public static void main(String args[]) throws Exception{
        Runtime r = Runtime.getRuntime();
        System.out.println("Total Memory: "+r.totalMemory());
        System.out.println("Free Memory: "+r.freeMemory());

        for(int i=0;i<1000000;i++){
            new MemoryTest();
        }

        System.out.println("After creating 1000000 instance, Free Memory: "+r.freeMemory());
        System.gc();
        System.out.println("After gc(), Free Memory: "+r.freeMemory());
    }
}
```

ACTIVIDAD CLASE RUNTIME

4. Ejecutar estadísticas de memoria JVM: [SystemMemory](#) disponible en el aula virtual

SHUTDOWN HOOKS PARA LA JVM EN JAVA

Veremos diferentes formas en que una aplicación JVM puede finalizar. Luego, usaremos las API de Java para administrar los “[shutdown hooks](#)” de cierre de JVM.

La JVM se puede [cerrar](#) de dos maneras diferentes:

- Un proceso controlado
- Una manera abrupta

SHUTDOWN HOOKS PARA LA JVM EN JAVA

Un proceso controlado [apaga la JVM](#) cuando:

- El último hilo que no es demonio termina. Por ejemplo, cuando el hilo principal sale, la JVM inicia su proceso de apagado.
- Envía de una señal de interrupción desde el sistema operativo. Por ejemplo, presionando Ctrl + C o cerrando sesión en el sistema operativo
- Llama a `System.exit()` desde código Java

SHUTDOWN HOOKS PARA LA JVM EN JAVA

Si bien todos nos esforzamos por lograr apagados elegantes, a veces la JVM puede cerrarse de manera abrupta e inesperada. La JVM se **apaga abruptamente** cuando :

- Envía de una señal de muerte desde el sistema operativo. Por ejemplo, emitiendo un `kill -9 <jvm_pid>`
- Llama a `Runtime.getRuntime().halt()` desde código Java
- El sistema operativo se apaga inesperadamente, por ejemplo, en caso de un corte de energía o de un pánico en el sistema operativo.

SHUTDOWN HOOKS PARA LA JVM EN JAVA

La JVM permite que se ejecuten funciones de registro antes de completar su apagado. Estas funciones suelen ser un buen lugar para liberar recursos u otras tareas domésticas similares. En la terminología de JVM, estas funciones se denominan **ganchos de apagado (shutdown hooks)**.

Los **ganchos de apagado** son básicamente subprocesos inicializados pero no iniciados . Cuando la JVM comience su proceso de apagado, iniciará todos los enlaces registrados en un orden no especificado. Después de ejecutar todos los ganchos, la JVM se detendrá.

SHUTDOWN HOOKS PARA LA JVM EN JAVA

La clase `Runtime` tiene dos métodos `addShutdownHook` y `removeShutdownHook`

Los **shutdown hooks** son una construcción especial que permite a los desarrolladores conectar un fragmento de código para que se ejecute cuando la JVM se está cerrando.

Esto es útil en casos en los que necesitamos realizar operaciones especiales de limpieza en caso de que la JVM se esté cerrando.

<https://www.geeksforgeeks.org/jvm-shutdown-hook-java/>

SHUTDOWN HOOKS PARA LA JVM EN JAVA

Usar un shutdown hook es directo.

Todo lo que tenemos que hacer es simplemente escribir una clase que herede de la clase `Thread` y proporcionar la lógica que queremos realizar cuando la JVM se está cerrando, dentro del método `public void run()`.

Luego registramos una instancia de esta clase como un gancho de apagado para la JVM llamando al método `Runtime.getRuntime().addShutdownHook(Thread)`.

Si necesita eliminar un shutdown hook previamente registrado, la clase `Runtime` también proporciona el método `removeShutdownHook(Thread)`.

SHUTDOWN HOOKS PARA LA JVM EN JAVA

Advertencia

La JVM ejecuta ganchos de apagado sólo en caso de terminaciones normales. Entonces, cuando una fuerza externa mata abruptamente el proceso de JVM, la JVM no tendrá la oportunidad de ejecutar ganchos de apagado. Además, detener la JVM desde el código Java también tendrá el mismo efecto. Por ejemplo,

```
Runtime.getRuntime().halt(129);
```

El método de detención finaliza por la fuerza la JVM que se está ejecutando actualmente. Por lo tanto, los ganchos de apagado registrados no tendrán la oportunidad de ejecutarse.

Este apartado es a título informativo puesto que veremos hilos(threads) en la próxima unidad didáctica.

ACTIVIDAD. SHUTDOWN HOOKS

5. Busca un ejemplo de shutdown hook útil en Java

CLASE PROCESS

API - Process class Java 11

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 JVM y devuelven un `objeto Java de la clase Process` que puede ser utilizado para controlar dicho proceso.

El proceso que crea otro proceso se llama proceso **padre** y el proceso creado proceso **hijo o subprocesso**.

En breve veremos la clase `ProcessBuilder`

ACTIVIDAD CLASE PROCESS

6. Crea un programa `RuntimeProcess` que cree un proceso mediante la clase `Runtime` para después destruirlo (el nombre del proceso se pasa por la línea de comandos)

ACTIVIDAD CLASE PROCESS

```
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");
            return;
        }

        try {
            Runtime runtime = Runtime.getRuntime();
            Process process = runtime.exec(args);
            Thread.sleep(1000); //Pausa la ejecución del hilo actual en los milisegundos especificados
            process.destroy();
        } catch (IOException ex) {
            System.err.println("Excepción de E/S!!");
        } catch (InterruptedException ex) {
            System.err.println("Excepción InterruptedException");
        }
    }
}
```

ACTIVIDAD CLASE PROCESS

Ejecuta el programa RuntimeProcess con la clase Mayusculas2

```
public class Mayusculas2 {

    public static void main(String[] args) {
        String str;
        str = args[0];
        System.out.println(str.toUpperCase());
    }
}
```

Ejemplo de ejecución desde la línea de comandos `> java RuntimeProcess java Mayusculas2 Gabriela`

LA CLASE PROCESS

Cuando se ejecuta un programa desde otro programa java, su salida por pantalla ya no sale. **¿Qué está pasando?.**



LA CLASE PROCESS

Queremos ejecutar un programa desde otro programa pero también capturar su salida...

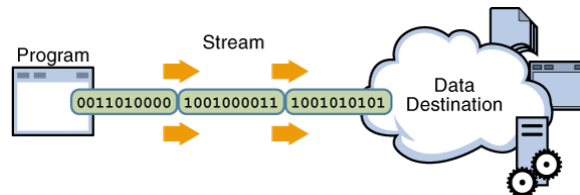
○ enviar datos a su entrada...



FLUJOS O STREAMS

Un **flujo (stream)** no es más que un objeto que hace de intermediario entre el programa y el origen o el destino de la información.

Es un canal de comunicación que un programa tiene con el mundo exterior.



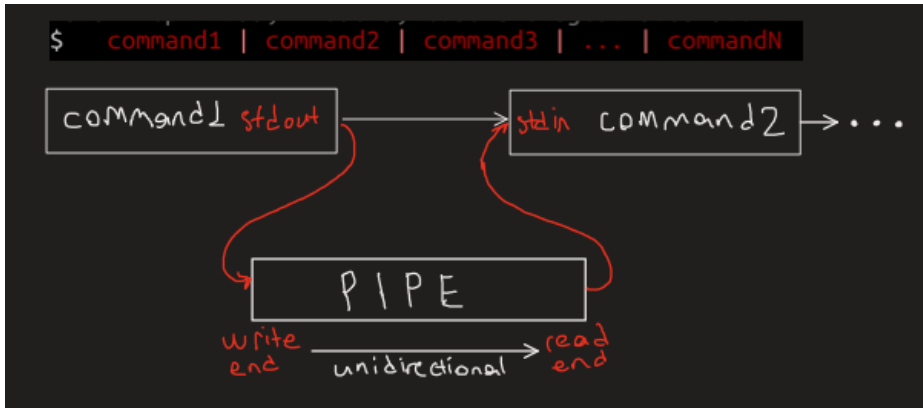
Este es un ejemplo de un programa usa un flujo de salida para escribir datos en un destino, un elemento a la vez.

LA CLASE PROCESS

La clase **Process** proporciona métodos para realizar entrada al proceso, obtener la salida del proceso, esperar a que se complete el proceso, verificar el estado de salida del proceso y destruir (matar) el proceso. Entre otros métodos tiene:

- **InputStream** `getInputStream()`
Devuelve el flujo de **entrada** conectado al la **salida** normal del proceso
- **OutputStream** `getOutputStream()`
Devuelve el flujo de **salida** conectado a la **entrada** normal del proceso
- **InputStream** `getErrorStream()`
Devuelve el flujo de **entrada** conectado a la **salida** de error del proceso

TUBERÍAS (PIPES)



TUBERÍAS

En los sistemas operativos de ordenadores tipo Unix, una tubería es una **secuencia de procesos encadenados** entre sí por sus flujos estándar, de modo que la salida de cada proceso (stdout) alimenta directamente la entrada (stdin) del siguiente.

La sintaxis de shell estándar para las tuberías es listar múltiples comandos, separados por barras verticales, | ("pipes" en la jerga común de Unix).

TUBERÍAS

Por ejemplo, para listar archivos en el directorio actual (`ls`), pero solo las líneas de salida de `ls` que contienen la cadena "key" (`grep`), y ver el resultado en una página desplazable (`less`), un usuario escribe lo siguiente en el línea de comando de un terminal:

```
ls -l | grep key | less
```

"`ls -l`" produce un proceso, cuya salida (`stdout`) se canaliza a la entrada (`stdin`) del proceso "`grep key`", y del mismo modo para el proceso de "`less`".

TUBERÍAS

Cada proceso toma información del proceso anterior y produce resultados para el siguiente proceso a través de flujos estándar.

```
ls -l | grep key | less
```

Cada "`|`" (pipe) indica al shell que conecte la salida estándar del comando de la izquierda a la entrada estándar del comando de la derecha mediante un mecanismo de comunicación entre procesos llamado un `pipe` (tubería), implementado en el sistema operativo.

Las tuberías son unidireccionales; los datos fluyen a través de la tubería de izquierda a derecha.

COMUNICACIÓN DE PROCESOS

Es importante recordar que un **proceso** es un programa en ejecución y, como cualquier programa, **recibe información, la transforma y produce resultados**.

COMUNICACIÓN DE PROCESOS

Esta acción se gestiona a través de:

La entrada estándar (stdin)

Lugar de donde el proceso lee los datos de entrada que requiere para su ejecución. No se refiere a los parámetros de ejecución del programa. Normalmente suele ser el **teclado**, pero podría recibirlos de un fichero, de la tarjeta de red o hasta de otro proceso, entre otros sitios.

La lectura de datos a lo largo de un programa (por ejemplo mediante `scanf` en C) leerá los datos de su entrada estándar.

COMUNICACIÓN DE PROCESOS

La salida estándar (stdout)

Lugar donde el proceso escribe los resultados que obtiene. Normalmente es la pantalla, aunque podría ser, entre otros, la impresora o hasta otro proceso que necesite esos datos como entrada. La escritura de datos que se realice en un programa (por ejemplo mediante printf en C o System.out.println() en Java) se produce por la salida estándar.

COMUNICACIÓN DE PROCESOS

La salida de error (stderr)

Lugar donde el proceso envía los mensajes de error. Habitualmente es el mismo que la salida estándar, pero puede especificarse que sea otro lugar, por ejemplo un fichero para identificar más fácilmente los errores que ocurren durante la ejecución.

La utilización de System.out y System.err en Java se puede ver como un ejemplo de utilización de estas salidas

COMUNICACIÓN DE PROCESOS

En Java, en cambio, el proceso hijo creado, objeto de la clase `Process` no tiene su propia interfaz de comunicación (terminal o consola) 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 y se puede acceder a ellas a través de los métodos de la clase `Process`:

- `getOutputStream()`: Devuelve el flujo de salida conectado a la entrada normal del subprocesso. El proceso puede escribir información en este flujo.

Es decir, este flujo está conectado por un pipe a la entrada estándar del subprocesso

COMUNICACIÓN DE PROCESOS

`getInputStream()`: Devuelve el flujo de entrada conectado a la salida normal del subprocesso. Es decir, este flujo está conectado por un pipe a la salida estándar del subprocesso

`getErrorStream()`: Devuelve el flujo de entrada conectado a la salida de error del subprocesso. Sin embargo, hay que saber que, por defecto, para la JVM, stderr está conectado al mismo sitio que stdout.

Utilizando estos flujos, el proceso padre puede enviarle datos al proceso hijo (subprocesso) y recibir los resultados de salida que este genere comprobando los errores.

COMUNICACIÓN DE PROCESOS

Hay que tener en cuenta que en algunos sistemas operativos, el tamaño de los buffers de entrada y salida que corresponde a stdin y stdout es limitado.

En este sentido, un fallo al leer o escribir en los flujos de entrada o salida del proceso hijo puede provocar que el proceso hijo se bloquee. Por eso, en Java se suele realizar la comunicación padre-hijo a través de un [buffer](#) utilizando los flujos vistos.

ACTIVIDAD CLASE PROCESS

7. Copia, analiza y ejecuta la clase [EjemploRunTimeExec](#)

Nota: **bash** (Bourne-again shell) es un programa informático, cuya función consiste en interpretar órdenes. Es un shell (intérprete de comandos) de Unix.

Ejecutar con argumentos el shell, con la opción `-c`.
Ejemplo: `bash -c "ls -l"`

```
package udl;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.BufferedReader;

public class EjemploRuntimeExec {

    Run | Debug
    public static void main(String args[]) {

        //Windows:
        //String commandLine="cmd /c dir";
        String [] commandLine= {"cmd", "/c", "dir"};

        //Linux (comando 'ls' SIN argumentos): String commandLine= "bash -c ls";
        //Linux (comando 'ls' CON argumentos):
        //String [] commandLine={"bash", "-c", "ls /"};
        //String [] commandLine={"bash", "-c", "java HolaMundo"};

        Process p=null;
        try {
            p = Runtime.getRuntime().exec(commandLine);
        } catch (IOException e) {
            e.printStackTrace();
        }

        try {
            (BufferedReader bri = new BufferedReader(new InputStreamReader(p.getInputStream())));
            (BufferedReader bre = new BufferedReader(new InputStreamReader(p.getErrorStream())));
            String line;

            while ((line = bri.readLine()) != null)
                System.out.println(line);

            while ((line = bre.readLine()) != null)
                System.out.println(line);

            p.waitFor();

            System.out.println("Done.");
        } catch (IOException | InterruptedException err) {
            err.printStackTrace();
        }
    }
}
```

ABRIENDO Y CERRANDO FLUJOS

Es una buena práctica cerrar explícitamente los flujos de E/S, ejecutando `close()` en la cláusula **finally** del bloque **try-catch-finally** para liberar los recursos del sistema inmediatamente cuando la transmisión ya no se necesita. Esto podría evitar graves fugas de recursos (resource leaks).

Lamentablemente, el método `close()` también arroja una `IOException`, y debe incluirse en una declaración **try-catch** anidada. Esto hace que los códigos sean algo feos.

```
InputStream is = null;
OutputStream os = null;
try {
    is = new FileInputStream("../input/fxrates.txt");
    os = new FileOutputStream("../output/fxrates.txt");
    // código que lee del flujo de entrada y escribe en el flujo de salida
} catch (...) {...}
finally {
    try { if (is != null) is.close(); } catch(IOException e) { //closing quietly }
    try { if (os != null) os.close(); } catch(IOException e) { //closing quietly }
}
```

CERRANDO FLUJOS: TRY-WITH-RESOURCES

Existe una alternativa **mucho mejor** si utilizas Java 7 en adelante.

A partir de esta versión de proporcionan declaraciones **try-with-resources** para la gestión automática de recursos en Java.

CERRANDO FLUJOS: TRY-WITH-RESOURCES

Todos los recursos abiertos en el **bloque try** serán **cerrados automáticamente** por Java, siempre que implementen las interfaces **Closable** y **AutoClosable**.

Dado que todos los `InputStream` y `OutputStream` son elegibles para ser utilizados dentro de las sentencias `try-with-resources`, **debes** tomar ventaja de esto.

Esto es muy bueno para los programadores de Java, ya que no son tan cuidadosos como sus homólogos de C++, especialmente cuando liberan recursos.

CERRANDO FLUJOS: TRY-WITH-RESOURCES

Este es el aspecto del código anterior con la sentencia try-with-resources

```
try (FileInputStream fis = new FileInputStream("../input/fxrates.txt");
    FileOutputStream fos = new FileOutputStream("../output/fxrates.tx")) {

    // código de lectura/escritura .....

} catch (IOException ioex) {
    System.out.println("Error en la copia de archivos : " + ioex.getMessage());
    ioex.printStackTrace();
}
```

ACTIVIDAD

8. Copia y ejecuta el programa `Mayusculas`
9. Escribe un programa que ejecute `Mayusculas` y le envía a su entrada estándar una cadena y lea de su salida estándar el resultado.

```
package udi;
import java.io.IOException;

public class Mayusculas {

    Run | Debug
    public static void main(String[] args) {
        byte[] entrada = new byte[100];

        try {
            System.out.println(x:"I'm 'Mayúsculas': Waiting for receiving some bytes through standard 'in'");
            System.in.read(entrada);
            //Convert the bytes to String, get the upper case String, and get bytes again
            entrada=(new String(entrada)).toUpperCase().getBytes();
            System.out.println(x:"I'm 'Mayúsculas': Bytes received, writing these bytes on standard 'out':");
            System.out.write(entrada);
        } catch (IOException ioe) {
            ioe.getMessage();
        }
    }
}
```

ACTIVIDAD

Recordar que la clase `Process` tiene entre otros los siguientes métodos:

- `InputStream getErrorStream()`

Devuelve el flujo de `entrada` conectado a la `salida` de error del subproceso

- `InputStream getInputStream()`

Devuelve el flujo de `entrada` conectado al la `salida` normal del subproceso

- `OutputStream getOutputStream()`

Devuelve el flujo de `salida` conectado a la `entrada` normal del subproceso

ACTIVIDAD

10. Redacta un programa llamado `'InvertirCadena'` que lea por la entrada estándar una cadena y muestre por la salida estándar la cadena invertida; por ejemplo:

```
'hola a todos'--->'sodot a aloh'
```

A continuación redactar otro programa llamado `'InvertirCadenas'` que le pida al usuario varias cadenas, hasta que el usuario introduzca la cadena `'FIN'`, y muestre esa misma cadena invertida. Para ello, el proceso `'InvertirCadenas'` debe lanzar un subproceso `'InvertirCadena'` para cada cadena a invertir.

CLASE PROCESSBUILDER

Java dispone en el paquete `java.lang` de varias clases para la gestión de procesos. En particular [ProcessBuilder](#)

[API – ProcessBuilder class Java SE 11](#)

Esta clase es una de las fundamentales para crear procesos del sistema operativo

CLASE PROCESSBUILDER

Cada instancia `ProcessBuilder` [gestiona](#) una colección de [atributos del proceso](#).

El método [start\(\)](#) de la clase [ProcessBuilder](#) crea un objeto [Process](#) con los atributos del `ProcessBuilder`. Este método puede ser invocado repetidamente para crear nuevos subprocesos con atributos idénticos o relacionados

CLASE PROCESSBUILDER

Por ejemplo, para ejecutar el comando `dir` de windows usando estas dos clases escribimos lo siguiente, indicando en el constructor de `ProcessBuilder` los argumentos del proceso que se quiere ejecutar como una lista de cadenas separadas por comas y luego utilizamos el método `start()` para iniciar el proceso.

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

Nota: `CMD` inicia una nueva instancia del intérprete de comandos de Windows. `/c` significa que ejecuta el comando especificado como cadena y luego termina

CLASE PROCESSBUILDER

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

- Un comando. Una lista de cadenas que representa el comando que se invoca y sus argumentos si los hay.
- Un entorno (environment) con sus variables (mapeo de variables a valores). El valor inicial es la copia del entorno del proceso en curso.
- Un directorio de trabajo. El valor por defecto es el directorio de trabajo del proceso en curso.

CLASE PROCESSBUILDER. DIRECTORIO DE TRABAJO

A veces puede ser útil cambiar el directorio de trabajo. En nuestro siguiente ejemplo vamos a ver cómo hacer precisamente eso.

El método `directory(File directory)` establece el directorio de trabajo del `ProcessBuilder`. Los subprocesos iniciados posteriormente por el método `start()` del objeto lo usarán como su directorio de trabajo.

```
import java.io.File;
import java.io.IOException;
public class ProcessBuilderDemo {

    public static void main(String[] args) {

        // create a new list of arguments for our process
        String[] list = {"notepad", "archivo1.txt"};

        // create the process builder
        ProcessBuilder pb = new ProcessBuilder(list);

        // set the working directory of the process
        pb.directory(new File("C:\\"));
        //pb.directory(new File(System.getProperty("user.dir")));
        System.out.println(" " + pb.directory());

        try{
            Process p = pb.start();
        } catch (IOException e){
            e.printStackTrace();
        }
    }
}
```

CLASE PROCESSBUILDER. ENTORNO

Método `environment()`

Map `environment()`

Este método devuelve un mapa de cadenas del entorno del `ProcessBuilder`.

Cada vez que se crea un `ProcessBuilder`, el entorno se inicializa a una copia del entorno de proceso actual. Los subprocesos iniciados posteriormente por el método `start()` del objeto utilizarán este map como su entorno.

VARIABLES DE ENTORNO

Las **variables de entorno** forman un conjunto de valores dinámicos que normalmente afectan al comportamiento de los procesos en un ordenador.

Por ejemplo, en Windows para mostrar la variable de entorno PATH (ruta de programas en el sistema) o la variable de entorno CLASSPATH (ruta de las clases del usuario)

```
echo %PATH%
```

```
echo %CLASSPATH%
```

En Linux,

```
echo $PATH
```

```
Echo $CLASSPATH
```

CLASE PROCESSBUILDER. ENTORNO

Antes de ver como modificamos el entorno, veamos que tipo de información podemos encontrar en el entorno default.

```
ProcessBuilder processBuilder = new ProcessBuilder();
Map<String, String> environment = processBuilder.environment();
environment.forEach((key, value) -> System.out.println(key + value));
```

Esto simplemente imprime cada una de las entradas de variables que se proporcionan por defecto.

```
ProcessBuilder pb = new ProcessBuilder(...)
Map<String, String> env = pb.environment();
env.put("MiVariableEntorno", "UnValor");
Process p = pb.start();
```

CLASE PROCESSBUILDER. ENTORNO

Para añadir una variable de entorno a nuestro objeto `ProcessBuilder`:

```
ProcessBuilder pb = new ProcessBuilder(...)  
Map<String, String> env = pb.environment();  
env.put("MiVariableEntorno", "UnValor");  
Process p = pb.start();
```

ACTIVIDAD

11. Copia, analiza y ejecuta la clase `RecibeDelSubproceso` con el programa `HolaMundo`

Este es un ejemplo que ejecuta un programa como `HolaMundo` y usa el método `getInputStream()` de la clase `Process` para obtener un flujo de entrada conectado a la salida del subproceso y poder así leer lo que el programa `HolaMundo` envía a la consola

```

import java.io.IOException;
import java.io.InputStream;
import java.io.InputStreamReader;
import java.util.Arrays;

//Recibe datos de la salida del subprocesso
public class RecibeDelSubproceso {

    public static void main(String args[]){

        if (args.length != 1){
            System.err.println("Se necesita un programa para ejecutar");
            System.exit(0);
        }
        BufferedReader br = null;

        try {
            Process process = new ProcessBuilder("java", args[0]).start();
            br = new BufferedReader(new InputStreamReader(process.getInputStream()));

            String line;

            System.out.println("Salida del proceso: " + args[0] + ": ");

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

        } catch (IOException e){
            e.printStackTrace();
        } finally {
            try{
                if (br != null) br.close();
            } catch (IOException e){
                e.printStackTrace();
            }
        }
    }
}

```

EJEMPLO

Este es otro ejemplo de cómo podemos ejecutar un comando de Linux, por ejemplo, cal 2023, capturar la salida de un subprocesso en un flujo de entrada y mostrarla.

```

import java.io.BufferedReader;
import java.io.InputStreamReader;

public class RecibeDelSubprocesoII {

    public static void main(String[] args) {

        System.out.println
            ("*****Calendar for Year*****");

        try {
            ProcessBuilder pb = new
                ProcessBuilder("cal", "2022");
            final Process p=pb.start();
            BufferedReader br=new BufferedReader(
                new InputStreamReader(
                    p.getInputStream()));
            String line;
            while((line=br.readLine())!=null){
                System.out.println(line);
            }
        } catch (Exception ex) {
            System.out.println(ex);
        }
        System.out.println
            ("*****");
    }
}

```


EJEMPLO

Supongamos ahora que queremos [ejecutar un proceso que necesita información de entrada](#). Si por ejemplo ejecutamos `date` desde la línea de comandos y pulsamos la tecla [Intro] nos pide escribir una nueva fecha

```
C:\Users\gabri>date
La fecha actual es: 02/10/2018
Escriba la nueva fecha: (dd-mm-aa) 03/10/2018
```

EJEMPLO

La clase `Process` posee el método [getOutputStream\(\)](#) que nos permite escribir en el flujo de entrada del subprocesso, así podemos enviarle datos.

El siguiente ejemplo ejecuta el comando `date` y le da los valores 03-10-2018. Con el método [write\(\)](#) se envían bytes al flujo, el método [getBytes\(\)](#) codifica la cadena en una secuencia de bytes que utilizan el juego de caracteres por defecto de la plataforma.

ACTIVIDAD

12) Copia, analiza y ejecuta la clase

[EnviaAlSubproceso](#)

```
import java.io.*;

//Recibe datos de la salida del subproceso
public class EnviaAlSubproceso {

    public static void main(String args[]){

        OutputStream os = null;
        BufferedReader br = null;

        try {
            Process process = new ProcessBuilder("CMD", "/c", "date").start();
            os = process.getOutputStream();

            os.write("01-10-18".getBytes());
            os.flush(); //vacía el flujo de salida

            br = new BufferedReader(new InputStreamReader(process.getInputStream()));

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

            int exitVal;

            exitVal = process.waitFor();
            System.out.println("Valor de salida: " +exitVal);

        } catch (IOException e){
            e.printStackTrace();
        } catch (InterruptedException ex){
            ex.printStackTrace();
        } finally {
            try{
                if (os != null) os.close();
                if (br != null) br.close();
            } catch (IOException e){
                e.printStackTrace();
            }
        }
    }
}
```

USO DE FLUSH()

Nota:

Los recursos asignados al [OutputStream](#) se liberan con el método [close\(\)](#). Este método, además, garantiza que los datos que hayamos escrito en el flujo pero que aun no hayan sido enviados (a un archivo, por ejemplo) se manden a su destino.

En los casos en que queramos asegurarnos de que los datos han sido enviados, pero no queramos cerrar el flujo, podemos usar el método [flush\(\)](#), que vacía los buffers de salida.

ACTIVIDAD

~~13. Analiza y ejecuta [EjemploProcessBuilder](#) (disponible en el aula virtual) **NO**~~

ACTIVIDAD

14. Escribe un programa en Java que lea dos números desde la entrada y visualice su suma. Controlar que lo introducidos por teclado sean dos números. Haz otro programa java para ejecutar el anterior.

```
import java.io.*;
import java.util.Scanner;

public class Suma {

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

        Scanner sc = new Scanner(System.in);
        int n1 = sc.nextInt();
        int n2 = sc.nextInt();
        System.out.println(n1+n2);

    }
}
```

```
import java.io.*;
import java.util.Scanner;

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

        OutputStream os = null;
        BufferedReader bri = null;
        BufferedReader bre = null;
        Scanner sc = new Scanner(System.in);

        try {
            ProcessBuilder pb = new ProcessBuilder("java", "Suma");
            Process process = pb.start();

            int n1 = sc.nextInt();
            int n2 = sc.nextInt();

            os = process.getOutputStream();
            os.write(("n1+\n").getBytes());
            os.flush();
            os.write(("n2+\n").getBytes());
            os.flush();
            os.close();

            bri = new BufferedReader(new InputStreamReader(process.getInputStream()));
            bre = new BufferedReader(new InputStreamReader(process.getErrorStream()));

            String line;
            while ((line=bri.readLine()) != null)
                System.out.println(line);

            while ((line=bre.readLine()) != null)
                System.out.println(line);

            int exitVal;
            exitVal = process.waitFor();
            System.out.println("Valor de salida: " +exitVal);

        } catch (Exception ex) {
            System.out.println(ex);
        }
    }
}
```

PROCESSBUILDER Y CÓMO REDIRECCIONAR LA ENTRADA Y SALIDA DE LOS PROCESOS

La clase `ProcessBuilder` tiene varios métodos que son útiles para **redireccionar** la entrada y salida de procesos externos ejecutados desde una aplicación Java.

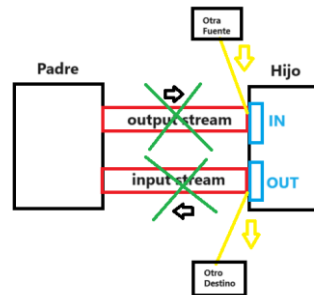
<code>ProcessBuilder.Redirect</code>	<code>redirectInput()</code> Returns this process builder's standard input source.
<code>ProcessBuilder</code>	<code>redirectInput(File file)</code> Sets this process builder's standard input source to a file.
<code>ProcessBuilder</code>	<code>redirectInput(ProcessBuilder.Redirect source)</code> Sets this process builder's standard input source.
<code>ProcessBuilder.Redirect</code>	<code>redirectOutput()</code> Returns this process builder's standard output destination.
<code>ProcessBuilder</code>	<code>redirectOutput(File file)</code> Sets this process builder's standard output destination to a file
<code>ProcessBuilder</code>	<code>redirectOutput(ProcessBuilder.Redirect destination)</code> Sets this process builder's standard output destination.

<http://tamanmohamed.blogspot.com/2012/06/jdk7-processbuilder-and-how-redirecting.html>

CLASE PROCESSBUILDER: REDIRECCIONES

El subprocesso lee la entrada de una tubería y el código java (padre) puede acceder a esta tubería por el flujo de salida retornado por `getOutputStream()`.

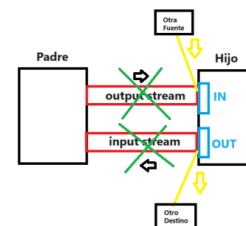
Sin embargo, la entrada estándar puede ser **redireccionada** desde otra fuente utilizando `redirectInput()`. En este caso `getOutputStream()` devolverá un flujo de salida nulo (null)



CLASE PROCESSBUILDER: REDIRECCIONES

Por defecto, el subprocesso escribe en las tuberías de salida y el error estándar. El código Java (padre) puede acceder a estas tuberías a través de los flujos de entrada devueltos por `getInputStream()` y `getErrorStream()`.

Igual que antes, la salida estándar y el error estándar pueden ser **redirigidos** a otros destinos utilizando `redirectOutput()` y `redirectError()`. En este caso, `getInputStream()` y/o `getErrorStream()` devuelven una secuencia de entrada nula (null)



PROCESSBUILDER Y CÓMO REDIRECCIONAR LA ENTRADA Y SALIDA DE LOS PROCESOS

La **clase anidada** `ProcessBuilder.Redirect` se ha introducido para proporcionar estas capacidades de redireccionamiento adicionales.

`ProcessBuilder.Redirect` proporciona otra forma de especificar cómo se redirigen los datos de E/S.

PROCESSBUILDER Y CÓMO REDIRECCIONAR LA ENTRADA Y SALIDA DE LOS PROCESOS

Cada instancia de `Redirect` es una de las siguientes

- el valor especial **`Redirect.PIPE`** (valor inicial)
- el valor especial **`Redirect.INHERIT`** (Indica que el origen o destino de E/S de subprocesso será el mismo que el del proceso actual)
- una redirección para leer desde un archivo, creada invocando a `Redirect.from(File)`
- una redirección para escribir en un archivo, creada invocando a `Redirect.to(File)`
- una redirección para añadir en un archivo, creada invocando a `Redirect.appendTo(File)`

ACTIVIDAD

15. Copia y ejecuta `ProcessBuilderRedirectDemo`

```
import java.io.File;
import java.io.IOException;
import java.lang.ProcessBuilder.Redirect;

public class ProcessBuilderRedirectDemo {
    public static void main(String[] args) throws IOException, InterruptedException {

        ProcessBuilder processBuilder = new ProcessBuilder("CMD", "/c", "dir");
        processBuilder.directory(new File("c:/dev"));

        File log = new File("c:/tmp/log_gg.txt");

        processBuilder.redirectErrorStream(true);
        processBuilder.redirectOutput(Redirect.to(log));

        Process p = processBuilder.start();
        p.waitFor();
        System.out.println("Done");
    }
}
```

ACTIVIDAD

16. Redacta un programa llamado 'invertirCadenas2' que realice la misma tarea que 'invertirCadenas' pero utilizando las clases `ProcessBuilder` y `ProcessBuilder.Redirect`
17. Redacta un programa que pida al usuario un nombre (o ruta) de un programa java y lance un subprocesso para ese programa utilizando la clase `ProcessBuilder`, estableciendo adecuadamente el valor de la variable de entorno '`CLASSPATH`'

Nota:

La variable `CLASSPATH` es la ruta que el entorno de ejecución de Java (JRE) busca las clases de los usuarios y otros archivos de recursos. Tanto el intérprete de Java como el compilador de Java utilizan `CLASSPATH` cuando buscan paquetes y clases de Java.

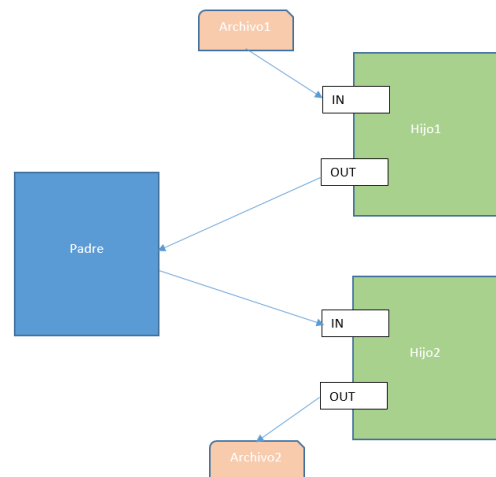
<https://www.oreilly.com/library/view/learning-java-4th/9781449372477/ch03s04.html>

ACTIVIDAD

18. a) Redactar las aplicaciones en Java denominadas 'Maximo', 'Mínimo' y 'Media', que calculen respectivamente el maximo, mínimo y media de una serie de números que reciban por la entrada estándar.
- b) Redactar una aplicación 'MaxMinMed' que reciba por su entrada estándar una lista de números y , lanzando los subprocesos 'Maximo', 'Mínimo' y 'Media' calcule esas 3 cantidades. El resultado se debe guardar en, respectivamente, los ficheros de texto 'Maximo.txt', 'Media.txt' 'Mínimo.txt'

ACTIVIDAD

19. Realizar un programa que reciba por la línea de comandos dos nombres de programa y dos nombres de archivo.
- El programa, proceso padre, creará procesos para los hijos (hijo1 e hijo2).
 - Conectará el primer archivo a la entrada del hijo1, y el segundo archivo a la salida del hijo2.
 - El programa recibirá la salida del hijo1 y la enviará a la entrada del hijo2.



ACTIVIDAD

20. Redactar un programa llamado '**Borrador**' al que se le pase un **nombre de archivo** y un **nombre de carpeta** por la **línea de comandos**. El objetivo es eliminar el archivo con ese nombre en la carpeta, o en cualquiera de sus subcarpetas (recursivamente). Ejemplo de invocación de 'Borrador':

```
$ java Borrador archivo.txt /home/usr/migue
```

A continuación redactar otro programa llamado '**BorradorCompleto**', al que se le pase por la **línea de comandos** un **nombre de archivo** y varias **nombres de subcarpetas**. El programa debe lanzar tantos subprocesos Borrador como carpetas se hayan indicado. Una vez lanzados los subprocesos el programa simplemente termina. Ejemplo de invocación de 'BorradorCompleto':

```
$ java BorradorCompleto archivo.txt /home/usr/migue /home/usr/pepe
```

Con este ejemplo se deben lanzar los siguientes subprocesos:

```
java Borrador archivo.txt /home/usr/migue
```

```
java Borrador archivo.txt /home/usr/pepe
```

REFERENCIAS

- Guide to [java.lang.ProcessBuilder API](#) (Baeldung)