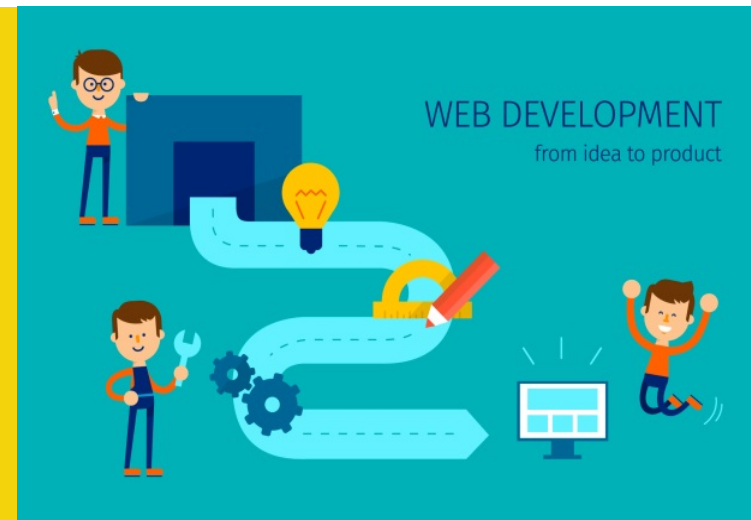


Módulo 9

Código: 490

Programación de Servicios y Procesos

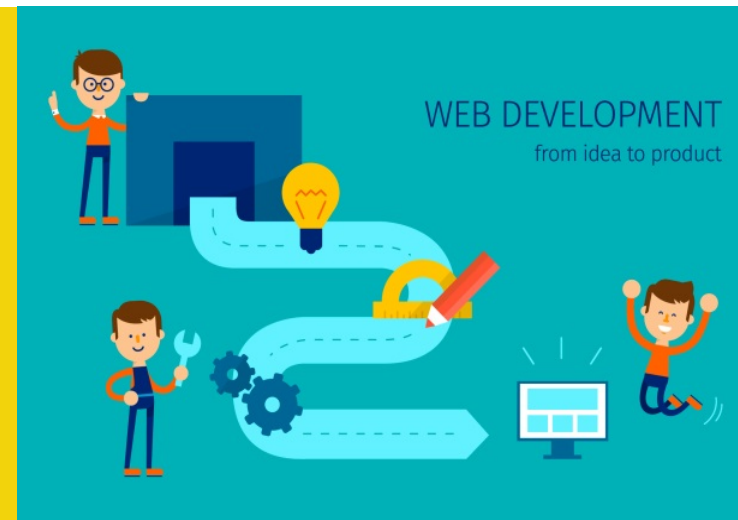
Técnico Superior en Desarrollo de
Aplicaciones Multiplataforma



UNIDAD 1

Programación multiproceso

Contenidos teóricos



1. Objetivos generales de la unidad
2. Competencias y contribución en la unidad
3. Contenidos conceptuales y procedimentales
4. Evaluación
5. Distribución de los contenidos
6. Sesiones

1. Objetivos generales de la unidad

Objetivos curriculares


1. Conocer las características de un proceso y su ejecución por el sistema operativo.

2. Conocer las características y diferencias entre programación concurrente, paralela y distribuida.

3. Utilizar Java para gestionar procesos en el sistema operativo.

4. Desarrollar programas que ejecuten tareas en paralelo.

2. Competencias y contribución en la unidad



Desarrollar aplicaciones multiproceso y multihilo empleando librerías y técnicas de programación específicas.

	A través de la gestión de los procesos del sistema operativo utilizando el lenguaje de programación Java.	
--	---	--

3. Contenidos

CONTENIDOS CONCEPTUALES		CONTENIDOS PROCEDIMENTALES
Procesos	<ul style="list-style-type: none">• Ejecutables, procesos y servicios.• Hilos.• Estados de un proceso.• Programación concurrente, paralela y distribuida.	<ul style="list-style-type: none">• Conocer los conceptos de ejecutable, proceso, servicio y hilo.• Ver las diferencias entre los conceptos anteriores.• Conocer los estados de un proceso.• Ver las características y diferencias existentes entre la programación concurrente, paralela y distribuida.
Gestión de procesos	<ul style="list-style-type: none">• Control de procesos en Windows.	<ul style="list-style-type: none">• Ver como se gestionan los procesos en el sistema operativo Windows.• Administrador de tareas.• Consola.

3. Contenidos

CONTENIDOS CONCEPTUALES		CONTENIDOS PROCEDIMENTALES
Trabajando con procesos en Java	<ul style="list-style-type: none">• Creación de procesos.• Comunicación entre procesos.• Sincronización de procesos.	<ul style="list-style-type: none">• Utilizar el API de Java para crear diferentes procesos desde programas.• Ver las formas de comunicar los diferentes procesos entre sí.• Ver los problemas inherentes a las programación concurrente y formas de solucionarlo.

4. Evaluación

Se han reconocido las características de la programación concurrente y sus ámbitos de aplicación.

Se han identificado las diferencias entre programación paralela y programación distribuida, sus ventajas e inconvenientes.

Se han analizado las características de los procesos y de su ejecución por el sistema operativo.

Se han caracterizado los hilos de ejecución y descrito su relación con los procesos.

Se han utilizado clases para programar aplicaciones que crean subprocesos.

Se han utilizado mecanismos para sincronizar y obtener el valor devuelto por los subprocesos iniciados.

4. Evaluación

Se han desarrollado aplicaciones que gestionen y utilicen procesos para la ejecución de varias tareas en paralelo.

Se han depurado y documentado los programas desarrollados.

5. Distribución de los contenidos

TEMPORALIZACIÓN DE LA UNIDAD

La unidad se compone de 6 sesiones, en las que se realizarán actividades prácticas distribuidas a lo largo de las diferentes sesiones como a continuación se indica:

- Sesión 1. ACTIVIDAD 1. Procesos y servicios.
- Sesión 1. ACTIVIDAD 2. Procesos y hilos.
- Sesión 1. ACTIVIDAD 3. Ventajas de la programación concurrente.
- Sesión 1. ACTIVIDAD 4. Planificación de procesos.
- Sesión 2. ACTIVIDAD 5. Administrador de tareas de Windows.
- Sesión 2. ACTIVIDAD 6. Gestión de procesos en Windows desde línea de comandos.
- Sesiones 3 y 4. ACTIVIDAD 7. Creación de procesos.
- Sesiones 3 y 4. ACTIVIDAD 8. Creación de procesos.
- Sesiones 3 y 4. ACTIVIDAD 9. Comunicación de procesos.
- Sesión 5. ACTIVIDAD 10. Comunicación de procesos.
- Sesión 5. ACTIVIDAD 11. Intercomunicación de procesos.

1. EJECUTABLES, PROCESOS Y SERVICIOS.

Ejecutable

- En informática, un **ejecutable** o **archivo ejecutable**, es un archivo binario, cuyo contenido se interpreta por el ordenador como un programa.
- Generalmente, contiene instrucciones en código máquina, pero también puede contener bytecode que requiera un intérprete para ejecutarlo.
- Dependiendo del tipo de instrucciones de que se traten, hablaremos de ejecutables portables (se pueden ejecutar en varias plataformas) y no portables (destinado a una plataforma concreta).
- Por ejemplo, un ejecutable Java es portable ya que utiliza un bytecode no asociado a un procesador en concreto.
- Sin embargo en un sentido más general, un programa ejecutable no tiene por qué necesariamente contener código de máquina, sino que puede tener instrucciones a interpretar por otro programa.

1. EJECUTABLES, PROCESOS Y SERVICIOS.

- Para determinar si un archivo es ejecutable los sistemas operativos se suelen basar en la extensión del archivo (normalmente .exe), aunque también sistemas operativos como UNIX lo hacen leyendo los bits de permiso de ejecución del archivo.



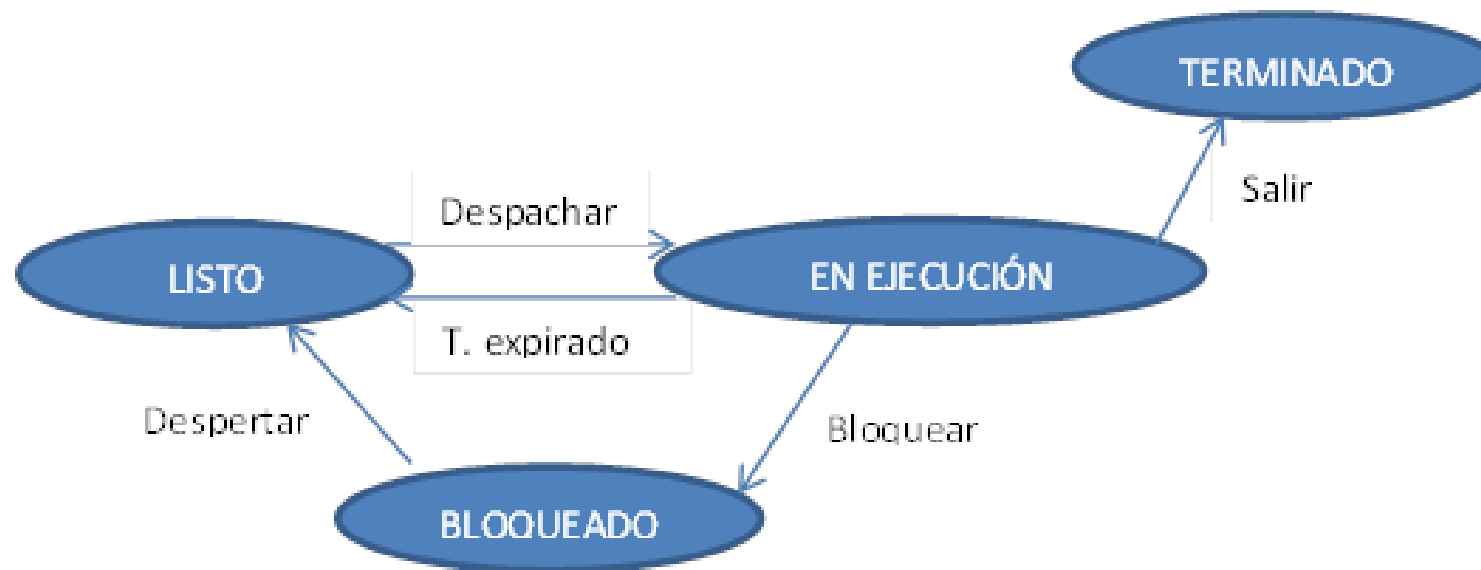
1. EJECUTABLES, PROCESOS Y SERVICIOS.

Procesos

- Un **proceso** es un programa que está en ejecución.
- Formalmente un proceso es *"Una unidad de actividad que se caracteriza por la ejecución de una secuencia de instrucciones, un estado actual, y un conjunto de recursos del sistema asociados"*.
- Durante su ciclo de vida el proceso puede pasar por los siguientes estados:
 - ✓ **En ejecución:** está usando el microprocesador.
 - ✓ **Listo:** el proceso está esperando a ser atendido por el microprocesador para continuar su ejecución.
 - ✓ **Bloqueado:** el proceso está esperando que ocurra un evento para continuar su ejecución.
 - ✓ **Terminado:** el proceso ha finalizado su ejecución

1. EJECUTABLES, PROCESOS Y SERVICIOS.

- Diagrama de estados de un proceso:



1. EJECUTABLES, PROCESOS Y SERVICIOS.

- Las transiciones entre estados son las siguientes:
 - ✓ **Listo – En ejecución:** la CPU es asignada al proceso para ejecutarse.
 - ✓ **En ejecución - Listo:** el proceso ha agotado su tiempo de uso de la CPU pero todavía no ha terminado, por lo que espera que de nuevo le sea entregada la CPU para continuar su ejecución.
 - ✓ **En ejecución - Bloqueado:** el proceso durante su ejecución, tiene que realizar alguna operación de E/S o esperar que ocurra algún evento externo.
 - ✓ **Bloqueado – Listo:** el evento externo ha ocurrido con lo cual el proceso puede continuar su ejecución, pero para ello tiene que esperar que le sea otorgada la CPU.
 - ✓ **En ejecución - Terminado:** el proceso finaliza su ejecución.

1. EJECUTABLES, PROCESOS Y SERVICIOS.

Servicio

- Los programas y servicios son programas en ejecución, pero los primeros normalmente se ejecutan en primer plano y por tanto tenemos su control, mientras que los servicios se ejecutan en segundo plano y por tanto nosotros a nivel de usuario no debemos interactuar con ellos.
- Habitualmente, un **servicio** es un programa que atiende a otro programa.



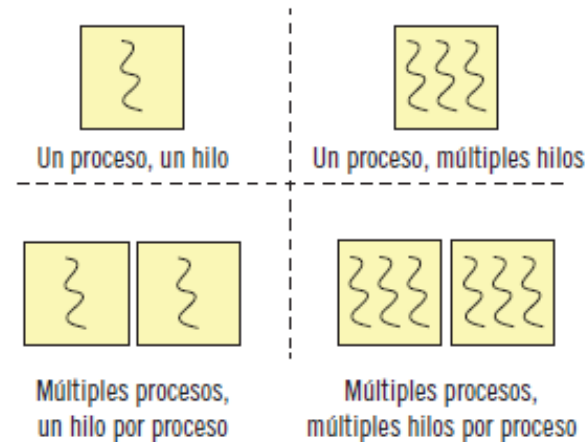
<http://figueredonorlan.blogspot.com/2016/11/sistemas-operativos-windows-y-linux.html>

2. HILOS

- Un hilo (thread), hebra, proceso ligero o subproceso es una secuencia de tareas encadenadas muy pequeña que puede ser ejecutada por un sistema operativo.
- Un hilo es simplemente una tarea que puede ser ejecutada al mismo tiempo que otra tarea.
- Los hilos que comparten los mismos recursos, junto con estos recursos, son los procesos.
- Un proceso sigue en ejecución mientras al menos uno de sus hilos siga activo. Cuando el proceso finaliza, todos sus hilos de ejecución también han terminado. Asimismo en el momento en el que todos los hilos de ejecución finalizan, el proceso no existe más y todos sus recursos son liberados.
- Algunos lenguajes de programación como Java tienen características que permiten a los programadores trabajar con hilos de ejecución.
- En el siguiente tema nos adentraremos en la forma de trabajar con hilos en Java.

2. HILOS

- En la siguiente imagen se puede ver la relación entre hilos y procesos:



- Los hilos comparten memoria, registros, CPU, etc.
- Es fundamental que se sincronicen para llevar a cabo su tarea, ya que sino se pueden producir efectos desagradables, como son los interbloqueos o las esperas indefinidas.

3. PROGRAMACIÓN CONCURRENTE

- La **programación concurrente** es una técnica de programación que permite la ejecución simultánea de varias tareas para resolver un problema.
- Estas tareas pueden ser procesos o hilos de ejecución.
- Las tareas se pueden ejecutar en una única CPU, en varios procesadores o en una red distribuida de ordenadores. Esto va a determinar que hablemos de **programación paralela y distribuida**.
 - A. **Programación paralela:**
 - ✓ Se utilizan varios procesadores (trabajando en paralelo) dentro de un mismo ordenador, para resolver el problema.
 - B. **Programación distribuida:**
 - ✓ Los procesadores utilizados están distribuidos geográficamente y se comunican a través de una red.

3. PROGRAMACIÓN CONCURRENTE

- Para entender el concepto de programación concurrente, hay que identificar los diferentes escenarios en los que se puede ejecutar un programa a nivel hardware:
 - ✓ **Entorno monoprocesador:**
 - Concurrencia virtual, ya que solo hay una CPU, por lo que solamente un proceso puede estar en un momento determinado en ejecución.
 - El sistema operativo se encarga de cambiar el proceso en ejecución después de un período corto de tiempo (del orden de milisegundos) creando en el usuario la percepción de que varios procesos se están ejecutando al mismo tiempo.

3. PROGRAMACIÓN CONCURRENTE

- ✓ Entorno multitarea, con varios núcleos en el procesador:
 - Cada núcleo podría estar ejecutando una instrucción diferente al mismo tiempo.
 - El sistema operativo se encarga de planificar los trabajos que se ejecutan en cada núcleo y cambiar unos por otros para generar multitarea.
 - Todos los cores comparten la misma memoria por lo que es posible utilizarlos de forma coordinada (*programación paralela*).
- ✓ Entorno distribuido:
 - Se dispone de varios ordenadores en red. Cada uno con sus propios procesadores y su propia memoria (*programación distribuida*).
 - Como cada ordenador posee su propia memoria, imposibilita que los procesos puedan comunicarse fácilmente, teniendo que utilizar otros esquemas de comunicación más complejos y costosos a través de la red.

3. PROGRAMACIÓN CONCURRENTES

- Una vez entendida la programación concurrente, y vistos los estados por los que puede pasar un proceso/hilo, nos puede surgir la pregunta de cómo se decide qué proceso pasa a la CPU y quién lo decide.
- La respuesta es, que el encargado de decidirlo es un componente del sistema operativo, y la política que sigue para elegir el proceso a ejecutar depende del **algoritmo de planificación de procesos** que siga.
- Los algoritmos utilizados son los siguientes:
 - ✓ **FIFO**: el proceso que primero llega a la espera será el que primero atienda la CPU.
 - ✓ **LIFO**: el último proceso que llegó será el primero en entrar en la CPU.
 - ✓ **Round Robin**: con este algoritmo se asigna un tiempo determinado de uso de la CPU denominado *quantum*. De tal forma que cuando un proceso agota su quantum pasa a la cola de espera, aplicándose en ella el algoritmo FIFO.

3. PROGRAMACIÓN CONCURRENTES

- ✓ **Prioridad estática:** a cada proceso se le asigna una prioridad, de tal forma que el sistema operativo va atendiendo a los procesos con una prioridad mayor.
- ✓ **Prioridad dinámica:** en este caso la prioridad de los procesos va cambiando, aumentando según va aumentando también el tiempo de espera. De esta forma, se evita que un proceso permanezca indefinidamente esperando a ser atendido por su prioridad.

4. GESTIÓN DE PROCESOS

- El sistema operativo es el encargado de crear los nuevos procesos a petición del usuario.
- La puesta en ejecución de un nuevo proceso se produce debido a que hay un proceso en concreto que está pidiendo su creación en su nombre o en nombre del usuario.
- Cualquier proceso en ejecución (*proceso hijo*) siempre depende del proceso que lo creó (*proceso padre*), estableciéndose un vínculo entre ambos.
- A su vez, el nuevo proceso puede crear nuevos procesos, formándose lo que se denomina un **árbol de procesos**.

4. GESTIÓN DE PROCESOS

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

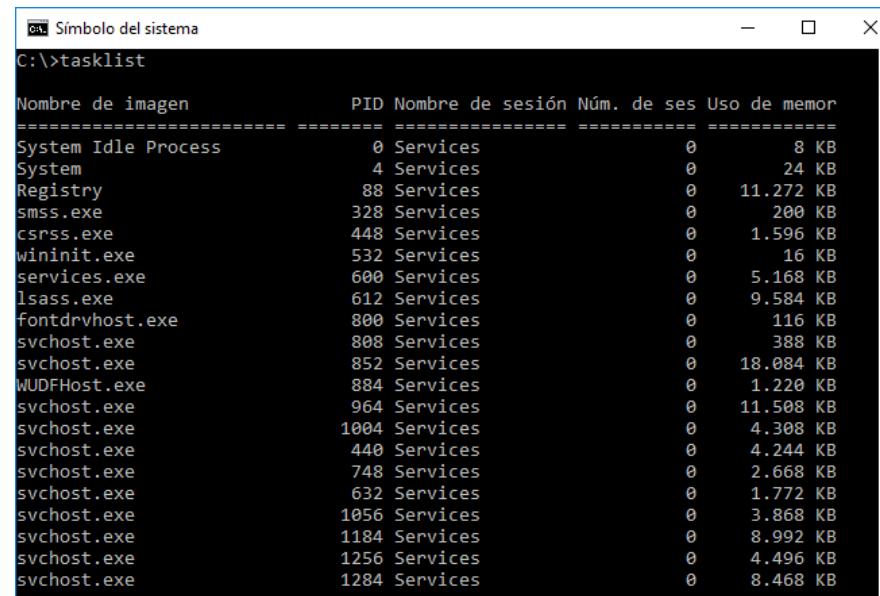
4. GESTIÓN DE PROCESOS

- La gestión de procesos se realiza de formas muy distintas en función de los dos grandes sistemas operativos: Windows y Linux.
- En Windows toda la gestión de procesos se realiza desde el **Administrador de tareas** al cual se accede con la combinación de teclas Ctrl + Alt + Supr:

Nombre	Estado	3% CPU	61% Memoria	10% Disco
Aplicaciones (9)				
Administrador de tareas		2,4%	21,5 MB	0 MB/s
Bloc de notas		0%	0,1 MB	0 MB/s
Bloc de notas		0%	0,7 MB	0 MB/s
Explorador de Windows		0%	28,4 MB	0 MB/s
Google Chrome (20)		0%	257,0 MB	0 MB/s
Herramienta Recortes		0%	0,3 MB	0 MB/s
Microsoft PowerPoint		0,9%	41,5 MB	0 MB/s
Microsoft Word (2)		0%	13,5 MB	0 MB/s
NetBeans IDE		0%	71,2 MB	0 MB/s
Procesos en segundo plano (61)				
64-bit Synaptics Pointing Enhance Service		0%	0,1 MB	0 MB/s
Adaptador de rendimiento inverso de WMI		0%	0,1 MB	0 MB/s
Adobe Acrobat Update Service (32 bits)		0%	0,1 MB	0 MB/s
Aplicación de subsistema de cola		0%	3,0 MB	0 MB/s

4. GESTIÓN DE PROCESOS

- También podemos usar comandos directamente en la consola CMD para administrar los procesos y tareas ejecutándose en nuestro equipo.
- Podemos con ellos obtener información y crear listas detalladas, detener aplicaciones, tareas y procesos aun cuando están bloqueados y no responden.



```
C:\>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         24 KB
Registry                        88 Services          0       11.272 KB
smss.exe                       328 Services          0         200 KB
csrss.exe                      448 Services          0        1.596 KB
wininit.exe                    532 Services          0          16 KB
services.exe                   600 Services          0         5.168 KB
lsass.exe                      612 Services          0         9.584 KB
fontdrvhost.exe                800 Services          0          116 KB
svchost.exe                    808 Services          0          388 KB
svchost.exe                    852 Services          0       18.084 KB
WUDFHost.exe                   884 Services          0          1.220 KB
svchost.exe                    964 Services          0       11.508 KB
svchost.exe                   1004 Services          0         4.308 KB
svchost.exe                    440 Services          0         4.244 KB
svchost.exe                    748 Services          0         2.668 KB
svchost.exe                    632 Services          0         1.772 KB
svchost.exe                   1056 Services          0         3.868 KB
svchost.exe                   1184 Services          0         8.992 KB
svchost.exe                   1256 Services          0         4.496 KB
svchost.exe                   1284 Services          0         8.468 KB
```

5. COMANDOS PARA LA GESTIÓN DE PROCESOS EN WINDOWS

- TASKLIST (listar tareas):
 - ✓ Muestra todas las aplicaciones ejecutándose en el equipo con el número de identidad del proceso (PID).
 - ✓ Si se usa sin opciones muestra una lista con el nombre del proceso, el PID (número de identidad del proceso) y la memoria usada.
 - ✓ Opciones:

TASKLIST /V	Muestra información detallada de cada tarea ejecutándose.
TASKLIST /SVC	Muestra información adicional de los servicios hospedados en cada proceso.
TASKLIST /M modulo	Muestra todas las tareas que usan un módulo DLL o EXE especificado. Si no se indica el modulo se muestran todos los módulos cargados.
TASKLIST /FO formato	Especifica el formato de salida. Puede ser: "TABLE", "LIST", "CSV".
TASKLIST /NH	Si se usan los formatos "TABLE" y "CSV", especifica que el "encabezado de columna" no se debe mostrar.
TASKLIST /FI filtro	Filtra la información que se solicita con un criterio especificado.

5. COMANDOS PARA LA GESTIÓN DE PROCESOS EN WINDOWS

- Ejemplos:

```
TASKLIST /V /FO CSV>%userprofile%/Desktop/list.csv
```

- Se crea en el escritorio una lista detallada de los procesos en ejecución en un archivo CSV.

```
TASKLIST /SVC /FO LIST>%userprofile%/Desktop/list.txt
```

- Se crea en el escritorio una lista sencilla de los procesos en ejecución con los servicios hospedados en cada proceso, en un archivo TXT.

5. COMANDOS PARA LA GESTIÓN DE PROCESOS EN WINDOWS

- TASKKILL (matar tareas):
 - ✓ Detiene ("mata") tareas o procesos usando su número de identidad del proceso (PID) o su nombre.
 - ✓ Opciones:

TASKKILL /PID identidadproceso	Detiene un proceso especificando el número de identidad. Este puede conocerse usando TASKLIST.
TASKKILL /IM nombreTarea	Detiene un proceso especificando su nombre.
TASKKILL /FI filtro	Permite usar un filtro para seleccionar varias tareas al mismo tiempo.
TASKKILL /T	Termina un proceso y todos los procesos secundarios iniciados por él.
TASKKILL /F	Especifica que se debe terminar un proceso de forma forzada.

5. COMANDOS PARA LA GESTIÓN DE PROCESOS EN WINDOWS

- Ejemplos:

`TASKKILL /F /IM notepad.exe`

- Finaliza el proceso del Bloc de notas de Windows.

`TASKKILL /PID 1250`

- Mata al proceso con el número de identidad 1250.

`TASKKILL /F /IM cmd.exe /T`

- Cierra la consola de CMD y todos los procesos secundarios iniciados por ella.

6. GÉSTIÓN DE PROCESOS EN JAVA

- Java dispone dentro del paquete `java.lang` de varias clases para la gestión de procesos.
- La clase que representa un proceso en Java es la clase `Process`.
- Las instancias de esta clase `Process` se gestionan utilizando los métodos de la clase `ProcessBuilder`.
- Antes de desgranar estas clases veamos un ejemplo:

```
public static void main(String[] args) {  
    try {  
        ProcessBuilder pb = new ProcessBuilder(  
            "C:\\Program Files (x86)\\Notepad++\\notepad++.exe");  
        Process p=pb.start();  
    } catch (IOException ex) {  
        Logger.getLogger(Ejemplo0.class.getName()).log(Level.SEVERE, null, ex);  
    }  
}
```

- En el ejemplo, lo que hace es abrir el bloc de notas.
- Para ello, en el constructor de `ProcessBuilder` se indican los argumentos del proceso que se quiere ejecutar, y posteriormente se llama al método `start()`, que lo que hace es iniciar el proceso.

6. GÉSTIÓN DE PROCESOS EN JAVA

- Métodos más importantes de la clase **Process**:

Método	Descripción
<code>InputStream getInputStream()</code>	Devuelve el flujo de entrada estándar del subprocesso. Permite leer lo que el subprocesso escriba en la consola.
<code>int waitFor()</code>	Hace que el proceso actual espere a que el subprocesso representado por el objeto <code>Process</code> finalice.
<code>InputStream getErrorStream()</code>	Devuelve el flujo de entrada conectado a la salida de error del subprocesso. Nos permite leer los errores producidos al lanzar el subprocesso.
<code>OutputStream getOutputStream()</code>	Devuelve el flujo de salida estándar del subprocesso. Permite escribir en el stream de entrada del subprocesso, para así poder comunicarnos con él.

6. GÉSTIÓN DE PROCESOS EN JAVA

Método	Descripción
<code>void destroy()</code>	Elimina el subproceso.
<code>int exitValue()</code>	Devuelve el valor de salida del subproceso.
<code>boolean isAlive()</code>	Comprueba si el proceso está vivo.

- El proceso (hijo/subproceso) que se crea con `Process`, no tiene su propia consola. Todas las operaciones de E/S se redirigen al proceso padre, donde se puede acceder a ellas usando los métodos mencionados anteriormente: `getInputStream`, `getOutputStream` y `getErrorStream`.

6. GÉSTIÓN DE PROCESOS EN JAVA

- Algunos de los métodos de la clase `ProcessBuilder` son:

Método	Descripción
<code>ProcessBuilder command()</code>	Indica los parámetros del proceso a ejecutar. Se puede indicar como una List o array de cadenas.
<code>Map<String,String> environment()</code>	Devuelve un mapa con las variables de entorno definidas.
<code>ProcessBuilder redirectError (File file)</code>	Redirecciona la salida de error a un fichero.
<code>ProcessBuilder redirectOutput (File file)</code>	Redirecciona la salida estándar a un fichero.
<code>ProcessBuilder redirectInput (File file)</code>	Establece la entrada estándar desde un fichero.
<code>ProcessBuilder directory (File directorio)</code>	Establece el directorio de trabajo.
<code>Process start()</code>	Inicia un nuevo proceso utilizando los atributos establecidos en el objeto <code>ProcessBuilder</code> .

6. GÉSTIÓN DE PROCESOS EN JAVA

- En el propio constructor del objeto **ProcessBuilder** podemos indicar:
 - ✓ El **comando** a ejecutar con sus parámetros si los tiene.
 - ✓ **Variables de entorno**.
 - ✓ **Directorio de trabajo**: por defecto, será el del proceso actual.
 - ✓ **Entrada estándar**: por defecto, el subprocesso lee la entrada de una tubería, a la que se puede acceder a través del método *getOutputStream()*. Aunque como hemos indicado en la tabla anterior, la entrada estándar puede ser redirigida a un fichero con el método *redirectInput*.
 - ✓ **Salida estándar**. Por defecto, el subprocesso escribe en la tuberías de salida estándar. Para acceder a ella utilizaremos el método *getInputStream()*. De la misma forma, esta salida puede ser redirigida a un fichero con el método *redirectOutput*.
 - ✓ **Salida de error**. Por defecto, el subprocesso escribe en la tuberías de error. Para acceder a ella utilizaremos el método *getErrorStream()*. Como en los casos anteriores, esta salida puede ser redirigida a un fichero con el método *redirectError*.

7. CREACIÓN DE PROCESOS EN JAVA

A continuación vemos algunos ejemplos de creación de subprocessos o procesos hijo:

```
public class Ejemplo1 {  
  
    public static void main(String[] args) throws IOException {  
  
        ProcessBuilder pb=new ProcessBuilder("cmd", "/c", "dir");  
        pb.start();  
  
    }  
}
```

- Este código ejecuta el comando DIR de MS-DOS en una consola de Windows.
- La consola la iniciamos con el comando CMD.
- A este comando pasamos el parámetro /C para que cuando ejecute el comando finalice.
- Al ejecutar no vemos ningún resultado porque el subprocesso envía el resultado del comando a la consola estándar.
- Si queremos recuperarlo para verlo, tendremos que usar el método `getInputStream()` para leer el stream de salida del proceso.

7. CREACIÓN DE PROCESOS EN JAVA

```
public class Ejemplo2 {  
  
    public static void main(String[] args) {  
  
        try {  
            ProcessBuilder ps = new ProcessBuilder();  
            ps.command("cmd", "/c", "dir");  
            Process p = ps.start();  
  
            BufferedReader br = new BufferedReader(  
                new InputStreamReader(p.getInputStream()));  
  
            String line;  
            while ((line = br.readLine()) != null) {  
                System.out.println(line);  
            }  
  
            int codigo = p.waitFor();  
            System.out.println("\nFinalizado con código de error: " + codigo);  
        } catch (InterruptedException | IOException ex) {  
            ex.printStackTrace();  
        }  
    }  
}
```

- Recogemos el stream de salida del proceso, y lo leemos utilizando un *InputStreamReader* con su buffer correspondiente.
- Leemos por líneas completas hasta que ya no haya nada más que leer.
- Con el método **waitFor()** hacemos que el proceso actual espere a que finalice el proceso hijo representado por el objeto *Process*.
- Este método recoge el código que devuelve **System.exit()** en Java. Que si todo ha ido bien será cero.
- Al ejecutar el código, mostrará un listado del directorio actual (el del proyecto), ya que no se ha indicado otro distinto con el método **directory()**.
- Un ejemplo de ejecución podría ser el siguiente:

7. CREACIÓN DE PROCESOS EN JAVA

```
Output - Procesos (run) x
run:
El volumen de la unidad C es SISTEMA
El número de serie del volumen es: 402B-F097

Directorio de C:\Users\profesor\Documents\NetBeansProjects\Procesos

24/04/2019 17:04 <DIR> .
24/04/2019 17:04 <DIR> ..
24/04/2019 17:04 <DIR> build
24/04/2019 16:38 3.609 build.xml
24/04/2019 16:38 85 manifest.mf
24/04/2019 16:38 <DIR> nbproject
24/04/2019 16:38 <DIR> src
      2 archivos 3.694 bytes
      5 dirs 107.197.947.904 bytes libres

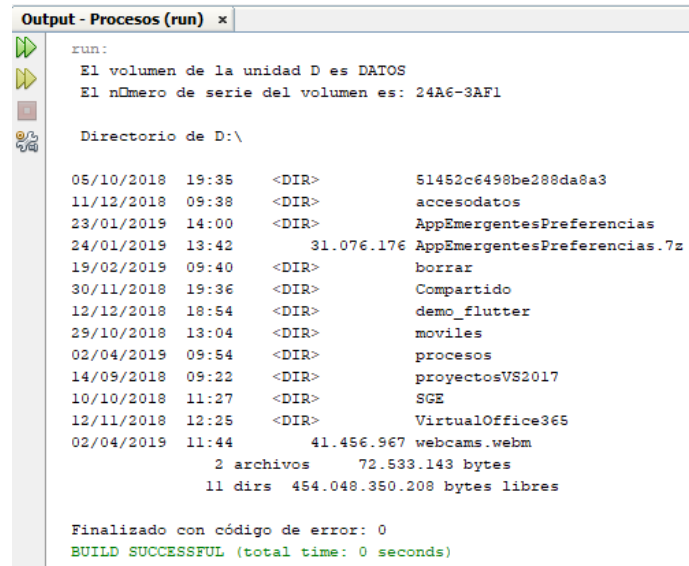
Finalizado con código de error: 0
BUILD SUCCESSFUL (total time: 0 seconds)
```

7. CREACIÓN DE PROCESOS EN JAVA

```
ProcessBuilder ps = new ProcessBuilder();  
ps.command("cmd", "/c", "dir");  
File directorio=new File("D:/");  
ps.directory(directorio);  
Process p = ps.start();
```

- Si quisiéramos cambiar el directorio en el que ejecutar el comando DIR, por ejemplo de la unidad D:, lo indicaríamos con un objeto File en el método *directory()* del ProcessBuilder.

Ahora el resultado es otro:



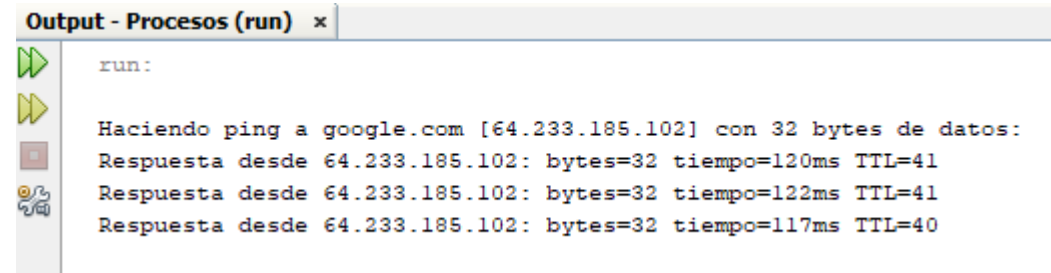
Output - Procesos (run) x

```
run:  
El volumen de la unidad D es DATOS  
El número de serie del volumen es: 24A6-3AF1  
  
Directorio de D:\  
  
05/10/2018 19:35 <DIR> 51452c6498be288da8a3  
11/12/2018 09:38 <DIR> accesodatos  
23/01/2019 14:00 <DIR> AppEmergentesPreferencias  
24/01/2019 13:42 31.076.176 AppEmergentesPreferencias.7z  
19/02/2019 09:40 <DIR> borrar  
30/11/2018 19:36 <DIR> Compartido  
12/12/2018 18:54 <DIR> demo_flutter  
29/10/2018 13:04 <DIR> moviles  
02/04/2019 09:54 <DIR> procesos  
14/09/2018 09:22 <DIR> proyectosVS2017  
10/10/2018 11:27 <DIR> SGE  
12/11/2018 12:25 <DIR> VirtualOffice365  
02/04/2019 11:44 41.456.967 webcams.webm  
2 archivos 72.533.143 bytes  
11 dirs 454.048.350.208 bytes libres  
  
Finalizado con código de error: 0  
BUILD SUCCESSFUL (total time: 0 seconds)
```


7. CREACIÓN DE PROCESOS EN JAVA

```
public class Ejemplo4 {  
  
    public static void main(String[] args) {  
  
        try {  
            ProcessBuilder ps = new ProcessBuilder();  
            ps.command("cmd.exe", "/c", "ping -n 3 google.com");  
            Process p = ps.start();  
  
            BufferedReader br = new BufferedReader(  
                new InputStreamReader(p.getInputStream()));  
  
            String line;  
            while ((line = br.readLine()) != null) {  
                System.out.println(line);  
            }  
  
            int codigo = p.waitFor();  
            System.out.println("\nFinalizado con código de error: " + codigo);  
        } catch (InterruptedException | IOException ex) {  
            ex.printStackTrace();  
        }  
    }  
}
```

- En este ejemplo hacemos un ping a Google y mostramos por consola de NetBeans el resultado.
- Con el parámetro "-n 3" indicamos a ping que haga 3 solicitudes de eco:



The screenshot shows the 'Output - Procesos (run)' window in NetBeans. It displays the output of the Java program, which is a ping command to google.com. The output shows three successful ping responses with varying times and TTL values.

```
run:  
  
Haciendo ping a google.com [64.233.185.102] con 32 bytes de datos:  
Respuesta desde 64.233.185.102: bytes=32 tiempo=120ms TTL=41  
Respuesta desde 64.233.185.102: bytes=32 tiempo=122ms TTL=41  
Respuesta desde 64.233.185.102: bytes=32 tiempo=117ms TTL=40
```

7. CREACIÓN DE PROCESOS EN JAVA

```
public class Ejemplo5 {  
  
    public static void main(String[] args) {  
  
        try {  
            ProcessBuilder ps = new ProcessBuilder();  
            ps.command("java", "com.everis.HolaMundo");  
            ps.directory(new File("./build/classes"));  
            Process p = ps.start();  
  
            BufferedReader br = new BufferedReader(  
                new InputStreamReader(p.getInputStream()));  
  
            String line;  
            while ((line = br.readLine()) != null) {  
                System.out.println(line);  
            }  
  
            int codigo = p.waitFor();  
            System.out.println("\nFinalizado con código de error: " + codigo);  
        } catch (InterruptedException | IOException ex) {  
            ex.printStackTrace();  
        }  
    }  
}
```

- En este otro ejemplo ejecutamos el programa Java "HolaMundo".
- El programa se encuentra en la carpeta *build* del proyecto, dentro del paquete *com.everis*, por lo que establecemos ese directorio con el método `directory()`.

8. COMUNICACIÓN DE PROCESOS EN JAVA

- Como sabemos, un **proceso padre** puede crear otros procesos llamado **subprocesos** o **procesos hijo**.
- A su vez, los hijos pueden crear nuevos hijos.
- Cuando se crea un nuevo proceso tenemos que saber que padre e hijo se ejecutan concurrentemente. Ambos procesos comparten la CPU y se irán intercambiando siguiendo la política de planificación del sistema operativo para proporcionar multiprogramación.
- En Java, el proceso hijo no tiene su propia interfaz de comunicación, por lo que el usuario no puede comunicarse con él directamente.
- Todas sus entradas y salidas de información se redirigen al proceso padre a través de streams:
 - ✓ **OutputStream**: flujo de salida del proceso hijo. El stream está conectado por una pipe (tubería) a la entrada estándar del proceso hijo.
 - ✓ **InputStream**: flujo de entrada del proceso hijo. El stream está conectado por una pipe a la salida estándar del proceso hijo.
 - ✓ **ErrorStream**: flujo de error del proceso hijo. El stream está conectado por una pipe a la salida estándar de errores del proceso hijo.

8. COMUNICACIÓN DE PROCESOS EN JAVA

- Utilizando estos streams, el proceso padre puede enviarle datos al proceso hijo y recibir los resultados de salida que éste genere comprobando los errores.
- Veamos un ejemplo:
 - ✓ La siguiente clase representa a un proceso que lo que hace es llenar un array con los números comprendidos entre 2 enteros recibidos como parámetros desde línea de comandos:

```
public class Proceso {  
  
    public void llenar(Integer a, Integer b) {  
        int[] array=new int[b-a+1];  
        for (int i = 0; i < array.length; i++) {  
            array[i]=a++;  
        }  
        System.out.println(Arrays.toString(array));  
    }  
  
    public static void main(String[] args) {  
        new Proceso().llenar(Integer.parseInt(args[0]),Integer.parseInt(args[1]));  
    }  
}
```

8. COMUNICACIÓN DE PROCESOS EN JAVA

- ✓ Ahora lo que vamos a hacer es un programa que representará al **proceso padre**, que cree varios de estos programas *Proceso*, que representarán a los **procesos hijos**. De esta forma se aprovechará al máximo la CPU para realizar la tarea de llenar varios arrays con enteros.

```
public class Lanzador {  
  
    public void crerProceso(Integer n1,  
        Integer n2, String fichResultado) {  
        try {  
            ProcessBuilder pb = new ProcessBuilder(  
                "java", "com.everis.Proceso",  
                n1.toString(),  
                n2.toString());  
            pb.directory(new File("./build/classes"));  
            pb.redirectError(new File("errores.txt"));  
            pb.redirectOutput(new File(fichResultado));  
            pb.start();  
        } catch (IOException e) {  
            // TODO Auto-generated catch block  
            e.printStackTrace();  
        }  
    }  
  
    public static void main(String[] args) {  
        Lanzador l = new Lanzador();  
        l.crerProceso(1, 5, "result1.txt");  
        l.crerProceso(6, 10, "result2.txt");  
    }  
}
```

8. COMUNICACIÓN DE PROCESOS EN JAVA

- Como vemos en el código anterior, el proceso padre crea 2 hijos.
- Cada uno de ellos se encarga de llenar un array.
- Además tanto la salida estándar como la salida de errores se redirecciona a ficheros, de forma que los resultados de la clase *Proceso* no se mostrarán por pantalla sino que se mandarán al fichero correspondiente.
- Asimismo, en caso de que ocurra cualquier excepción durante la ejecución de los procesos hijo, éstos mensajes se mandarán al fichero *errores.txt*.

9. SINCRONIZACIÓN DE PROCESOS

- Cuando los procesos (tanto padres como hijos) pueden acceder a recursos compartidos como por ejemplo variables, bases de datos o dispositivos, hay que tener cuidado si no se “controla” el orden en el acceso a estos recursos, ya que sino se pueden producir resultados inesperados e incluso errores.
- Es decir, es necesaria una **sincronización** del trabajo que realizan los diferentes procesos, en la que un proceso **espere** a que otro finalice su tarea para continuar él con la suya.
- Los mecanismos de comunicación entre procesos vistos anteriormente, se pueden considerar como métodos de sincronización ya que permiten al proceso padre llevar el ritmo de envío y recepción de mensajes.
- Así, si por ejemplo el proceso padre necesita leer de la salida del hijo a través de su *InputStream*, éste se bloquea hasta que el hijo le proporcione los datos requeridos. En este sentido, padre e hijo se están sincronizando, ya que el padre “espera” al hijo.

9. SINCRONIZACIÓN DE PROCESOS

- Cuando hablamos de sincronización, aparece el concepto de **exclusión mutua**.
- La exclusión mutua, se refiere a impedir que 2 o más procesos lean o escriban simultáneamente datos compartidos.
- Así, de forma paralela se define el concepto de **región crítica** como la parte de un programa en la que se accede a un recurso compartido.
- Entonces al final, sincronizar procesos va a consistir en implementar mecanismos que garanticen la exclusión mutua entre procesos, cuando éstos accedan a regiones críticas de los programas.
- A parte de usar los streams, otros mecanismos de espera que se pueden usar para implementar la sincronización entre procesos son:
 - ✓ Semáforos.
 - ✓ Tuberías (pipes)
 - ✓ Colas de mensajes.
 - ✓ Bloques de memoria compartida.

9. SINCRONIZACIÓN DE PROCESOS

Semáforos

- Hablaremos de los semáforos, como mecanismo por excelencia para manejar la exclusión mutua. Básicamente un semáforo es una variable entera positiva sobre la que se pueden realizar 2 funciones:
 - ✓ P: reservar
 - ✓ V: liberar.
- Así por ejemplo si queremos controlar el acceso a una impresora, el semáforo se inicializa a 1 indicando que está libre. Cuando un proceso quiera usar la impresora ejecutará P sobre el semáforo y cuando deje de usarla ejecutará V sobre el semáforo.

9. SINCRONIZACIÓN DE PROCESOS

- En **Java** la forma de implementar la exclusión mutua es utilizando la palabra clave **synchronized** para indicar que comienza una sección crítica en el código del programa. De esta forma la máquina virtual de Java protege el código automáticamente si varios procesos intentan acceder a ella de forma simultánea.
- Todo eso se verá en detalle en la siguiente unidad didáctica.

```
/* La máquina virtual Java evitará que más de un proceso/hilo
acceda a este método*/
synchronized public void actualizarSaldo(double importe) {
    /*...*/
    this.saldo = importe;
}
```

10. PROGRAMACIÓN MULTIPROCESO

- Llegados al final de la unidad, tenemos ya claro que la programación concurrente es una forma eficaz de procesar la información al permitir que diferentes procesos se vayan alternando en la CPU para proporcionar multiprogramación.
- Es el sistema operativo el que se encarga de proporcionar multiprogramación entre todos los procesos del sistema, y de forma totalmente transparente a los usuarios y desarrolladores.
- Ahora bien, si lo que se pretende es realizar procesos **cooperantes**, que realicen tareas conjuntamente, será el desarrollador el responsable de implementar técnicas que permitan la comunicación y sincronización entre ellos.
- En este caso, podemos definir las siguientes **fases** a la hora de realizar un programa multiproceso cooperativo:
 1. Descomposición funcional.
 2. Partición.
 3. Implementación.

10. PROGRAMACIÓN MULTIPROCESO

1. Descomposición funcional.

- Identificación de las tareas que debe realizar la aplicación y las relaciones existentes entre ellas.

2. Partición.

- Distribución de las tareas en procesos estableciendo el esquema de comunicación entre ellos.
- Como son procesos cooperativos necesitarán información unos de otros por lo que deben comunicarse, con la consiguiente sincronización.

3. Implementación.

- Identificar la plataforma a usar, por ejemplo Java, y utilizar las herramientas disponibles con el objetivo de realizar la comunicación y sincronización de procesos para realizar la cooperación.