

# Programación Multiproceso

PSP - UT 1

2º DAM

# Índice

1. [Conceptos básicos](#)
2. [Programación concurrente, paralela y distribuida](#)
3. [Creación de Procesos](#)
4. [Comunicación de procesos](#)
5. [Gestión de procesos](#)
6. [Sincronización de procesos](#)
7. [Programación multiproceso](#)



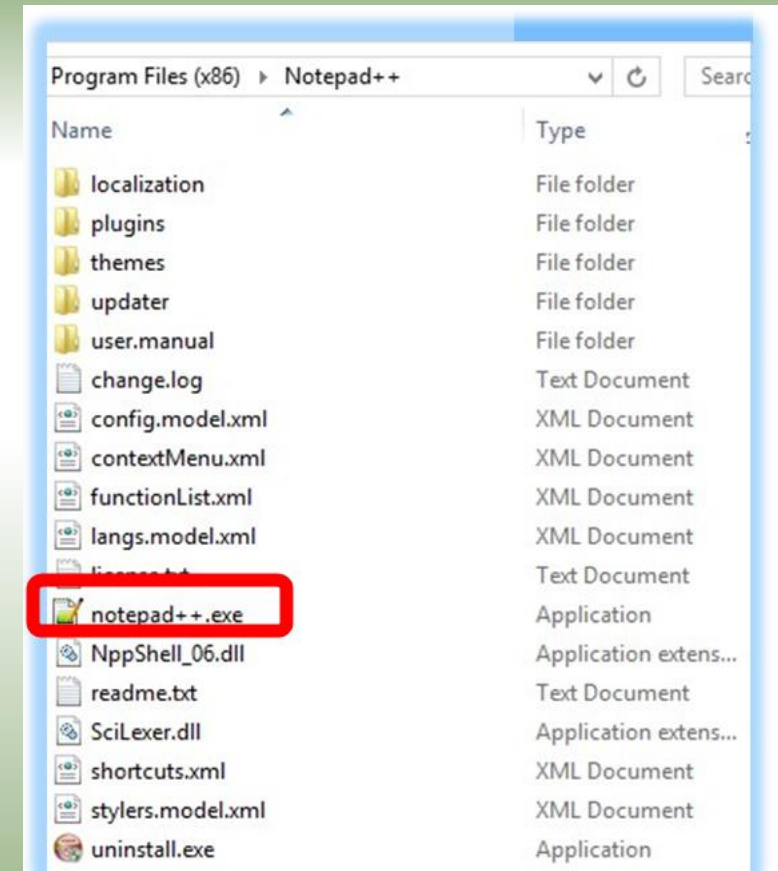
1

# Conceptos básicos

# Ejecutables

Un ejecutable es un archivo con la estructura necesaria para que el sistema operativo pueda iniciar el programa que hay dentro. En Windows, los ejecutables suelen ser archivos con la extensión **.EXE** (abreviatura de EXECUTABLE), pero también existen los archivos **.COM** (que fueron los primeros), los **.BAT** (archivos de procesamiento de lotes, que permitían encadenar la ejecución de varios programas sucesivamente), incluso algunas librerías como **.DLL** que contienen información adicional para el sistema operativo y pueden ser ejecutados por este.

Ejemplos de ejecutables en Linux son **.bin** **.run** **.py** o **.sh**, Los archivos **.bin** y los **.run** suelen ser instaladores de programas, mientras que los **.sh** son scripts que ejecutas directamente en la consola. En Linux, para que un fichero sea ejecutable, hay que darle permiso de ejecución.



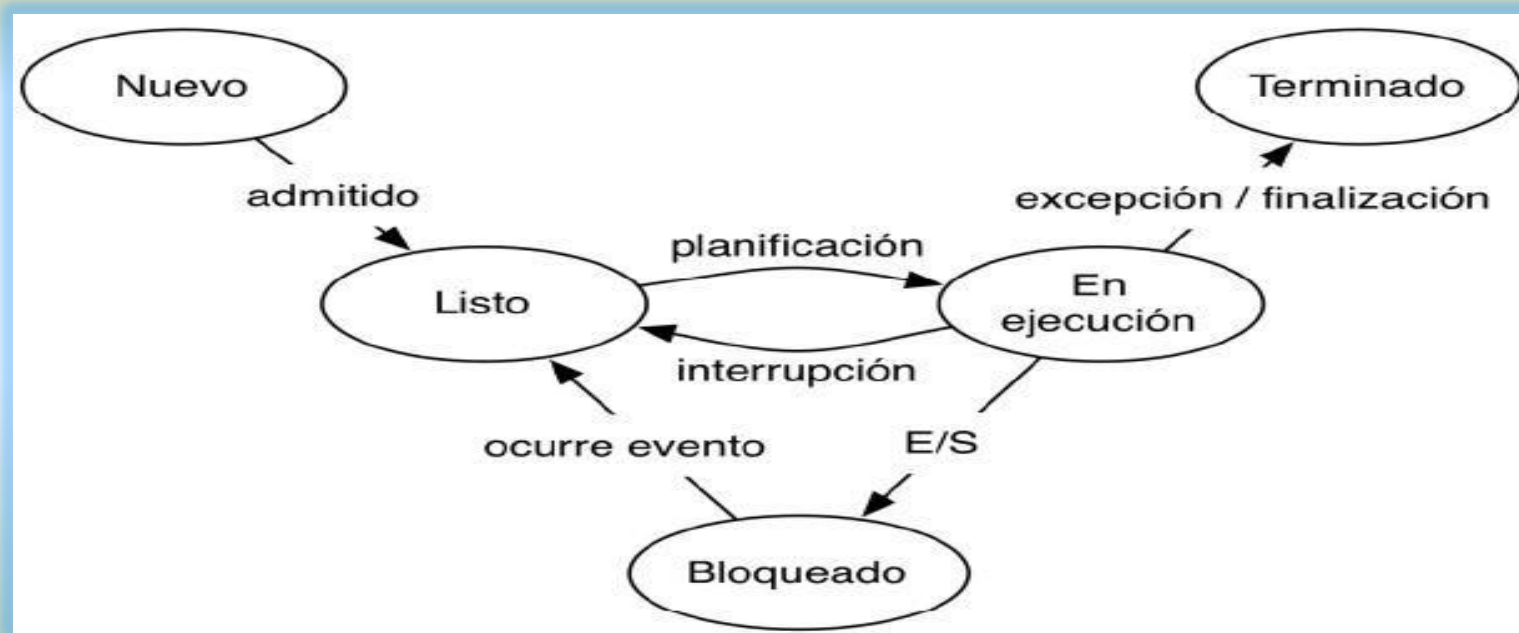
# Procesos

Cuando un programa se ejecuta, se crea un proceso que queda bajo el control del sistema operativo. Un proceso puede atravesar diversas etapas en su «ciclo de vida».

Los principales estados en los que puede estar un estado son:

- **Nuevo:** El proceso está siendo creado a partir del ejecutable
- **Listo:** El proceso no se encuentra en ejecución, pero está esperando para hacerlo. El planificador del sistema operativo no le ha asignado todavía un procesador
- **En ejecución:** El proceso se está ejecutando, está dentro del microprocesador. El sistema operativo utiliza el mecanismo de interrupciones para controlar su ejecución.
- **Bloqueado:** El proceso está bloqueado esperando que ocurra algún suceso
- **Terminado:** El proceso ha finalizado su ejecución y libera su imagen de memoria

Existen otros estados, pero ya son muy dependientes del sistema operativo concreto



Cada proceso se representa en el sistema operativo con un bloque de control de proceso (PCB, Process Control Block). En este PCB se guardan una serie de elementos de información de estos. Estos elementos son: el identificador del proceso, el estado del proceso, registros de CPU (acumuladores, punteros de la pila, registros índices y registros generales), información de planificación de CPU (prioridad de proceso, punteros a colas de planificación, etc.), información de gestión de memoria, información de contabilidad (tiempo de uso de CPU, números de procesos, etc.), información de estado de dispositivos E/S (lista de archivos abiertos, etc.), esta estructura es variable según el SSOO, pero en general suele estar compuesto por los siguientes componentes:

Name	Status	CPU	Memory	Disk	Network
▶  Notepad++ : a free (GNU) sourc...		0%	9,7 MB	0 MB/s	0 Mbps

## **Demonio**

Proceso no interactivo que está ejecutándose continuamente en segundo plano. Suele proporcionar un servicio básico a otros procesos.

## **Servicio**

Programa que atiende a otro programa. Es un proceso que no muestra ninguna ventana ni gráfico en pantalla porque no está pensado para que el usuario lo maneje directamente.



# Hilo

Un hilo es un proceso ligero, o subproceso que puede ser ejecutado por el sistema operativo. En JAVA, se usa la clase Thread para definir hilos de ejecución concurrentes dentro de un mismo proceso.

Un proceso ocupa su propio espacio de memoria, tal que no tiene acceso a los datos de otros procesos. Sin embargo, un hilo sí puede acceder a los datos de otro hilo del mismo proceso.

# Sistema operativo

Un sistema operativo se puede definir como la capa de software que equipa a las computadoras, cuya labor es la de administrar y gestionar todo el hardware que interactúa en la misma y proporcionar una interfaz sencilla a los programas para comunicarse con dicho hardware, permitiendo utilizar los recursos del computador de forma eficiente.





# Programación concurrente, paralela y distribuida

## Sistema monoproceso

- Permite la ejecución de un solo proceso
- Hasta que no finalice, ningún otro proceso podrá ser atendido

(Por ejemplo: MS-DOS)

## Sistema multiproceso

Permite la ejecución de varios procesos simultáneamente, en los que se intenta maximizar la utilización de la CPU, por lo que los procesos se ejecutan simultáneamente en la CPU y sólo quedan a la espera de ejecución si requieren de algún recurso del sistema que esté ocupado en ese momento, en cuanto obtiene dicho recurso, podrá ejecutarse de nuevo.

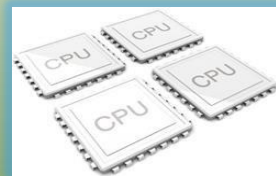
# Programación concurrente

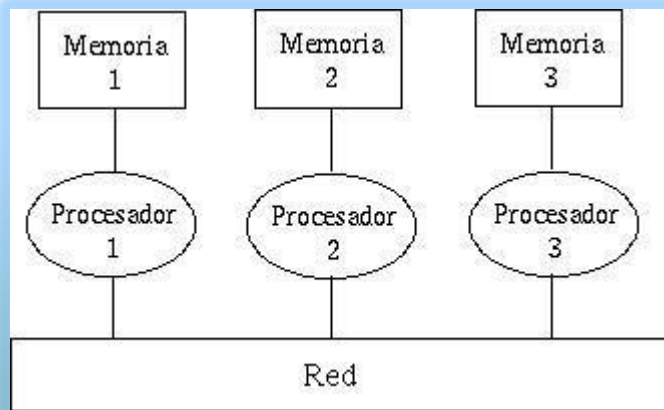
La programación concurrente es la parte de la programación que se ocupa de crear programas que pueden tener varios hilos que colaboran para ejecutar un trabajo y aprovechar al máximo el rendimiento de sistemas multiproceso. En el caso de la programación concurrente un solo ordenador puede ejecutar varias tareas a la vez (lo que supone que tiene 2 o más núcleos).

# Programación paralela y distribuida

Dentro de la programación concurrente tenemos la paralela y la distribuida:

- En general se denomina «programación paralela» a la capacidad de un ordenador (con varios núcleos o no) de ejecutar dos o más tareas a la vez, normalmente repartiendo el tiempo de proceso entre las tareas.
- Se denomina «programación distribuida» al software que se ejecuta en ordenadores distintos y que se comunican a través de una red. Se alcanzan elevadas mejoras de rendimiento. No se comparte memoria entre los procesadores, por lo que se utilizan esquemas complejos y costosos de comunicaciones a través de la red que los conecta.





Veamos un ejemplo, una película de cine se compone de una sucesión de imágenes estáticas -fotogramas-, que nos dan la sensación de movimiento al proyectarse a una velocidad de 24 imágenes por segundo. Películas como *Buscando a Nemo*, *Final Fantasy*, *Shrek*, *Toy Story*, etc., generadas totalmente por ordenador, necesitan que se "dibuje", renderice es el término exacto, cada uno de los fotogramas; para una película de 90 minutos, se necesita renderizar 129.600 fotogramas.

[Pixar](#), autora de algunas de estas películas, comenta en su web [3] que cada imagen les lleva renderizarla unas 6 horas, algunas hasta 19 horas; para esos 90 minutos harían falta 88 años de cálculo para que un sólo ordenador las completara, sin embargo, repartiendo el trabajo entre 200 ordenadores podría hacerse en tan sólo 6 meses.



# 3

## Creación de Procesos en JAVA



En Java es posible crear procesos utilizando algunas clases que el entorno ofrece para esta tarea.

Ejemplo1 de cómo ejecutar un proceso del S.O. con la clase **ProcessBuilder**.

```
public class LanzadorProcesos {  
  
    public void ejecutar(String ruta){  
  
        ProcessBuilder pb;  
  
        try {  
  
            pb = new ProcessBuilder(ruta);  
  
            pb.start();  
  
        } catch (Exception e) {  
  
            e.printStackTrace();  
  
        }  
  
    }  
  
    public static void main(String[] args) {  
  
        String ruta= "C:\\Program Files\\Notepad++\\notepad++.exe";  
  
        LanzadorProcesos lp=new LanzadorProcesos();  
  
        lp.ejecutar(ruta);  
  
        System.out.println("Finalizado");  
  
    }  
  
}
```

## ProcessBuilder y Process

- ✓ La clase **ProcessBuilder** es usada para ejecutar programas en un proceso separado de una forma simple.
- ✓ El método **start()** crea un nuevo proceso, que es devuelto en una instancia de la clase **Process**, usada para gestionar el proceso creado.
- ✓ Se puede invocar repetidamente para crear más procesos con los mismos o diferentes atributos

✓ Los procesos (**Process**) cuando finalizan devuelven el estado en el que terminaron. Para saber dicho estado, se puede invocar la función **exitValue**. Este estado sirve para determinar si el proceso terminó satisfactoriamente o bajo algún estado de error. Por convenio, el estado “0” significa que el proceso terminó correctamente. Cualquier otro estado determinará una situación de error

Ejemplo 2 de cómo ejecutar un proceso del S.O. con la clase **ProcessBuilder**.

Para completar este ejemplo, nos falta:

- Redireccionar la salida de error
- Cambiar el directorio de trabajo
- Leer los datos devueltos por el proceso.

```
public class Sumador {  
    public static int sumar(int n1, int n2){  
        return (n1 + n2);  
    }  
  
    public static void main(String[] args){  
        int n1=Integer.parseInt(args[0]);  
        int n2=Integer.parseInt(args[1]);  
        int suma= Sumador.sumar(n1, n2);  
        System.out.println(suma);  
    }  
}  
  
public class Lanzador {  
    public void lanzarSumador(int n1, int n2){  
        String clase="Sumador";  
        ProcessBuilder pb;  
        try {  
            pb = new ProcessBuilder("java", clase, String.valueOf(n1),  
                                    String.valueOf(n2));  
            pb.start();  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
    public static void main(String[] args){  
        Lanzador l=new Lanzador();  
        l.lanzarSumador(1, 51);  
        l.lanzarSumador(51, 100);  
    }  
}
```

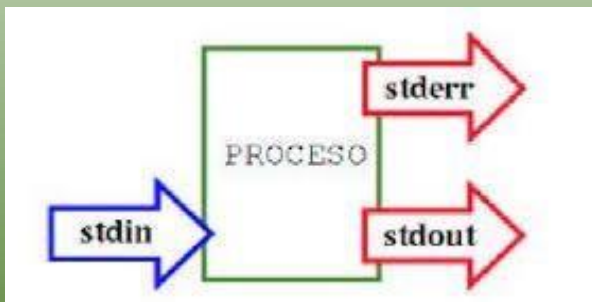


# 4

## Comunicación de procesos

Las operaciones multiproceso pueden implicar que sea necesario pasar información entre los procesos, lo que obliga a la necesidad de utilizar mecanismos específicos de comunicación que ofrecerá Java o a diseñar alguno separado que evite los problemas que puedan aparecer. Normalmente se hace uso de ficheros para establecer la comunicación entre los procesos.

El subproceso lee la información de la entrada estándar (stdin) que por defecto es el teclado, no hay que confundirlo con los parámetros de ejecución del programa, y escribe los resultados en la salida estándar (stdout) y los errores en la salida de errores estándar (stderr), que por defecto ambas son la pantalla. Es habitual configurar un fichero de log como salida estándar.



### Ejemplo 3: Mostrar datos generados por Sumador.

```
public class Lanzador {  
    public void lanzarSumador(int n1, int n2) throws IOException {  
        try {  
            ProcessBuilder pb = new ProcessBuilder("java", "Sumador", String.valueOf(n1),  
                                                    String.valueOf(n2));  
            // Cambia el directorio de trabajo, al directorio donde se encuentran los .class  
            pb.directory(new File("bin"));  
            pb.redirectError(new File("files" + File.separator + "error.log"));  
  
            pb.start();  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
  
        // Mostramos caracter a caracter la salida generada por Sumador  
        try (InputStream is = process.getInputStream()) {  
            int c;  
            while ((c = is.read()) != -1) {  
                System.out.print((char) c);  
            }  
        } catch (IOException e) {  
            e.printStackTrace();  
            throw e;  
        }  
    }  
}  
  
public static void main(String[] args){  
    Lanzador l=new Lanzador();  
    l.lanzarSumador(1, 51);  
}
```

Ejemplo 4: Escribir datos generados por Sumador en un fichero.

```
public class Lanzador {  
    public void lanzarSumador(int n1, int n2) throws IOException {  
        try {  
            ProcessBuilder pb = new ProcessBuilder("java", "Sumador", String.valueOf(n1),  
                                                    String.valueOf(n2));  
            // Cambia el directorio de trabajo, al directorio donde se encuentran los .class  
            pb.directory(new File("bin"));  
            pb.redirectError(new File("files" + File.separator + "error.log"));  
            pb.redirectOutput(new File("files" + File.separator + "salida.log"));  
  
            pb.start();  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
  
    }  
    public static void main(String[] args){  
        Lanzador l=new Lanzador();  
        l.lanzarSumador(1, 51);  
    }  
}
```



# Gestión de procesos

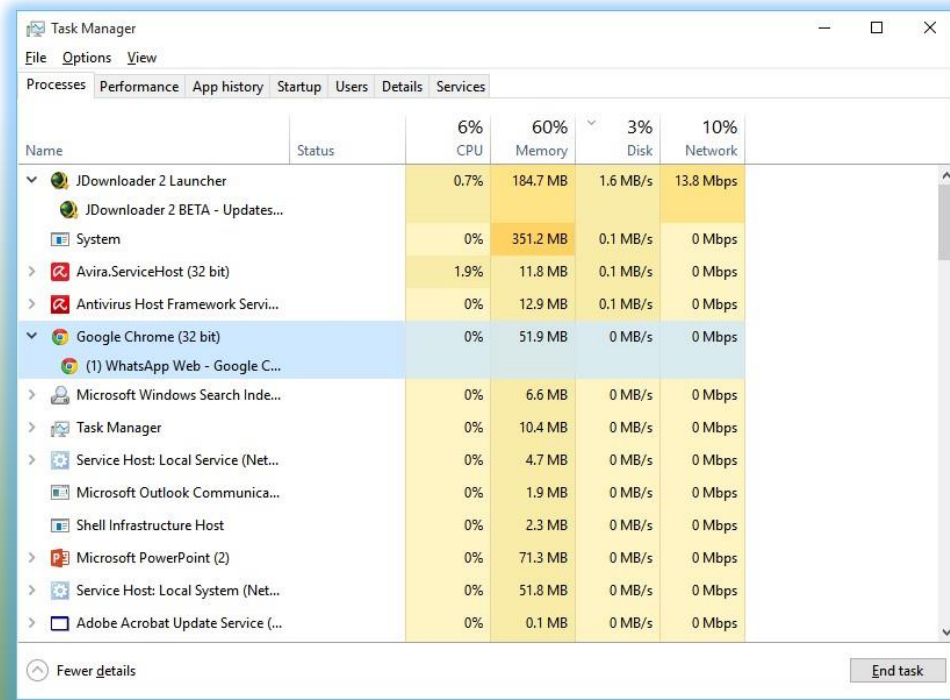


5

# Árbol de procesos

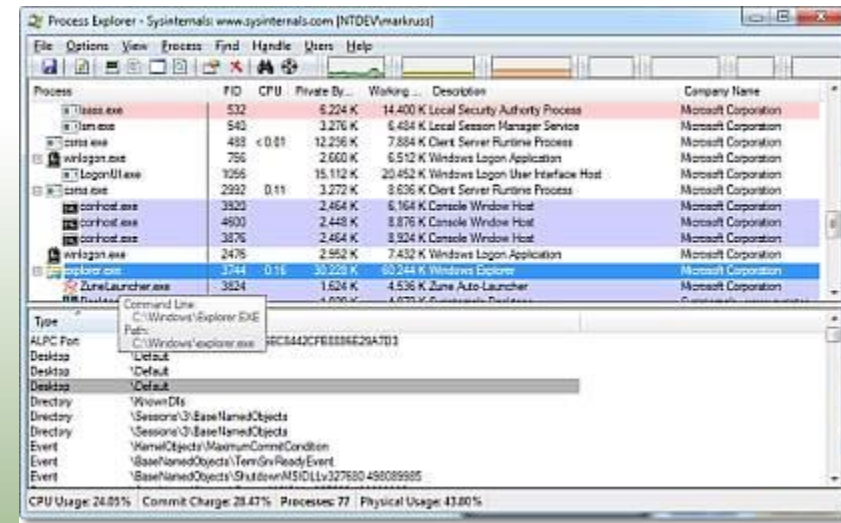
- ✓ El sistema operativo es el encargado de crear los nuevos procesos siguiendo las directrices del usuario
- ✓ La puesta en ejecución de un nuevo proceso se produce debido a que hay un proceso en concreto que está pidiendo su creación en su nombre o en nombre del usuario
- ✓ Cualquier proceso en ejecución (proceso hijo) normalmente depende del proceso que lo creó (proceso padre), estableciéndose un vínculo entre ambos
- ✓ El nuevo proceso hijo puede a su vez crear nuevos procesos, formándose lo que se denomina un árbol de procesos
- ✓ Los sistemas operativos proporcionan herramientas para visualizar los procesos en ejecución en diferentes vistas.

En Windows toda la gestión de procesos se realiza desde el «**Administrador de tareas**» al cual se accede con Ctrl+Alt+Supr. Existen otros programas más sofisticados que proporcionan algo más de información sobre los procesos, como **Process Explorer** (antes conocido con el nombre de ProcessViewer).



The screenshot shows the Windows Task Manager window with the 'Processes' tab selected. It lists various running applications and system processes, including JDownloader 2 Launcher, System, Avira.ServiceHost, Antivirus Host Framework Servi..., Google Chrome, Microsoft Windows Search Inde..., Task Manager, Service Host: Local Service (Net..., Microsoft Outlook Communica..., Shell Infrastructure Host, Microsoft PowerPoint (2), Service Host: Local System (Net..., and Adobe Acrobat Update Service (...). The columns show the Name, Status, CPU usage, Memory usage, Disk usage, and Network usage for each process.

Name	Status	6% CPU	60% Memory	3% Disk	10% Network
JDownloader 2 Launcher		0.7%	184.7 MB	1.6 MB/s	13.8 Mbps
JDownloader 2 BETA - Updates...					
System		0%	351.2 MB	0.1 MB/s	0 Mbps
Avira.ServiceHost (32 bit)		1.9%	11.8 MB	0.1 MB/s	0 Mbps
Antivirus Host Framework Servi...		0%	12.9 MB	0.1 MB/s	0 Mbps
Google Chrome (32 bit)		0%	51.9 MB	0 MB/s	0 Mbps
(1) WhatsApp Web - Google C...					
Microsoft Windows Search Inde...		0%	6.6 MB	0 MB/s	0 Mbps
Task Manager		0%	10.4 MB	0 MB/s	0 Mbps
Service Host: Local Service (Net...		0%	4.7 MB	0 MB/s	0 Mbps
Microsoft Outlook Communica...		0%	1.9 MB	0 MB/s	0 Mbps
Shell Infrastructure Host		0%	2.3 MB	0 MB/s	0 Mbps
Microsoft PowerPoint (2)		0%	71.3 MB	0 MB/s	0 Mbps
Service Host: Local System (Net...		0%	51.8 MB	0 MB/s	0 Mbps
Adobe Acrobat Update Service (...)		0%	0.1 MB	0 MB/s	0 Mbps

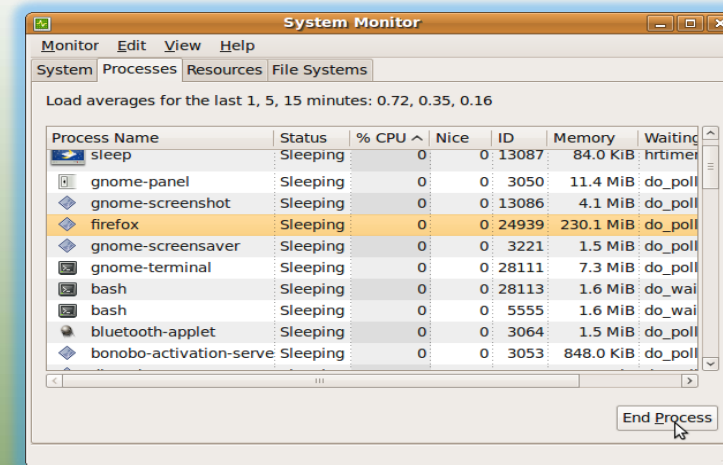


The screenshot shows the Process Explorer window from Sysinternals. It displays a detailed list of running processes, including their PID, CPU usage, Private Bytes, Working Set, Description, and Company Name. The processes listed include smss.exe, csrss.exe, winlogon.exe, csrss.exe, conhost.exe, explorer.exe, and ZuneLauncher.exe. The bottom status bar shows CPU Usage: 24.85%, Commit Charge: 28.47%, Processes: 77, and Physical Usage: 43.80%.

Process	PID	CPU	Private By...	Working ...	Description	Company Name
smss.exe	532		6,224 K	14,400 K	Local Security Authority Process	Microsoft Corporation
csrss.exe	540		3,276 K	6,464 K	Local Session Manager Service	Microsoft Corporation
winlogon.exe	488	< 0.01	12,256 K	7,884 K	Client Server Runtime Process	Microsoft Corporation
csrss.exe	756		2,600 K	6,512 K	Windows Logon Application	Microsoft Corporation
conhost.exe	1056		15,112 K	20,452 K	Windows Logon User Interface Host	Microsoft Corporation
csrss.exe	2992	0.11	3,272 K	8,636 K	Client Server Runtime Process	Microsoft Corporation
conhost.exe	3820		2,464 K	6,164 K	Console Window Host	Microsoft Corporation
conhost.exe	4900		2,448 K	8,876 K	Console Window Host	Microsoft Corporation
conhost.exe	3876		2,464 K	8,924 K	Console Window Host	Microsoft Corporation
winlogon.exe	2476		2,462 K	7,432 K	Windows Logon Application	Microsoft Corporation
explorer.exe	3744	0.16	30,228 K	60,244 K	Windows Explorer	Microsoft Corporation
ZuneLauncher.exe	3824		1,624 K	4,536 K	Zune Auto-Launcher	Microsoft Corporation

Los procesos en los sistemas UNIX están identificados por un número que es único (PID). Otra información importante es el número de identificación de usuario (UID) y el número de identificación del grupo de usuarios al que pertenece el proceso (GID). Se puede utilizar un **terminal de consola** para la gestión de procesos, lo que implica que no solo se pueden arrancar procesos, también podemos detenerlos, reanudarlos, terminarlos y modificar su prioridad de ejecución.

```
PID TTY          TIME CMD
6668 ?           00:00:02 gnome-terminal
6703 ?           00:00:00 \_ gnome-pty-helpe
6704 pts/0        00:00:00 \_ bash
7813 pts/0        00:00:00 \_ ps
6846 ?           00:00:41 gnome-system-mo
6936 ?           00:00:07 chrome
6943 ?           00:00:00 \_ chrome
6944 ?           00:00:00 \_ chrome-sandbox
6945 ?           00:00:01 \_ chrome
6949 ?           00:00:00 \_ nacl_helper
6953 ?           00:00:00 \_ chrome
7095 ?           00:00:01 \_ chrome
7562 ?           00:00:00 \_ chrome
6966 ?           00:00:01 wicd-client
```



## Comandos para la gestión de procesos en sistemas libres y propietarios.

### Creación de procesos

- ✓ Cuando se crea un nuevo proceso hijo, ambos procesos, padre e hijo se ejecutan concurrentemente, y comparten la CPU y se irán intercambiando siguiendo la política de la planificación del sistema operativo
- ✓ El proceso padre tiene la posibilidad de esperar hasta que el hijo termine su ejecución (*operación **wait***)
- ✓ Los procesos son independientes y tienen su propio espacio de memoria asignado, llamado imagen de memoria
- ✓ En sistemas Windows, no existen apenas comandos para gestionar procesos.

## Terminación de procesos

Al terminar la ejecución de un proceso, es necesario avisar al sistema operativo de su terminación para liberar los recursos que tenga asignados

Un proceso hijo puede terminar de dos formas

✓ Voluntaria: Realiza su ejecución completa liberando sus recursos al finalizar

✓ Forzosa: El proceso padre puede ordenar la terminación de un proceso hijo mediante el método **destroy()**. Esta operación elimina el proceso hijo indicado liberando sus recursos en el sistema operativo



# 6

## Sincronización de procesos

Cuando se lanza más de un proceso de una misma sección de código no se sabe qué proceso ejecutará qué instrucción en un cierto momento, lo que es muy peligroso.

Si dos o más procesos avanzan por esta sección de código es perfectamente posible que unas veces nuestro programa multiproceso se ejecute bien y otras no.

En todo programa multiproceso pueden encontrarse estas zonas de código «peligrosas» que deben protegerse especialmente utilizando ciertos mecanismos. El nombre global para todos los lenguajes es denominar a estos trozos «**secciones críticas**».



Los mecanismos más típicos son los ofrecidos por UNIX/Windows:

- Semáforos.
- Colas de mensajes.
- Tuberías (pipes)
- Bloques de memoria compartida.

En realidad, algunos de estos mecanismos se utilizan más para intercomunicar procesos, aunque para los programadores Java la forma de resolver el problema de la «sección crítica» es más simple. En Java, si el programador piensa que un trozo de código es peligroso puede ponerle la palabra clave **synchronized** y la máquina virtual Java protege el código automáticamente.

La palabra reservada **synchronized** se usa para indicar que ciertas partes del código, (habitualmente, una función miembro) están sincronizadas, es decir, que solamente un subproceso puede acceder a dicho método a la vez.

A continuación, se muestran unos ejemplos de uso del modificador **synchronized**

```
/* Sincronización de un método (a nivel de objeto) */
public synchronized void metodo1(){
    //...
}

/* Sincronización de un método estático (a nivel de clase)*/
public static synchronized void metodo2(){
    //...
}

/* Sincronización de un bloque de código (a nivel de clase)*/
private static Object _objCreationCriticalRegion=new Object();

private List<String> lColores=new ArrayList<>();

public void addColor(String color){

    synchronized (objCreationCriticalRegion ) {
        lColores.add(color);
    }
}
```

Debemos tener en cuenta que podemos crear bloqueos a nivel de clase o a nivel de objeto (ver SynchronizationExample y CuentaBancaria):

- **Bloqueo a nivel de objeto:** El bloqueo a nivel de objeto es provisto por un objeto concreto, por ejemplo, la instancia **this**. Se produce, por ejemplo, cuando sincronizamos un método no estático o un bloque de código no estático. Cuando un hilo tiene tomado el bloqueo, ningún otro hilo puede ejecutar ningún otro método `synchronized` del mismo objeto, es decir, se garantiza que se ejecutará con régimen de exclusión mutua respecto de otro método del mismo objeto que también sea `synchronized`.
- **Bloqueo a nivel de clase:** El bloqueo a nivel de clase es uno para cada clase y representando por el literal **class**, por ejemplo, **`String.class`**. Evita que varios hilos entren en un bloque sincronizado durante la ejecución en cualquiera de las instancias disponibles de la clase. Esto significa que, si hay 100 instancias de la clase durante la ejecución de un programa, entonces solo un hilo en ese momento podrá ejecutarse, y todos los demás casos serán bloqueados para otros hilos. La sincronización a nivel de clase garantiza que dos hilos no puedan ejecutar un método sincronizado al mismo tiempo o en paralelo, por lo que se hace en exclusión mútua con cualquier otro método estático `synchronized` de la misma clase.

El patrón de diseño único **Singleton** lo usamos para garantizar que una clase solo se pueda instanciar una vez. Debemos garantizar que tenemos un punto de acceso global a ella. Útil en caso de operaciones costosas, o controlar accesos a un mismo recurso a la vez.

```
public class Singleton {
    //crear referencia como estática
    private static Singleton instance = null;

    private static int numInstancias = 0;

    // Poner el constructor privado para que no se puedan crear más instancias
    private Singleton(){
        numInstancias++;
    }

    // creador sincronizado para protegerse de posibles problemas multi-hilo
    // garantizar que solo habrá una única instancia
    private synchronized static void createInstance() {
        if (instance == null) {
            instance = new Singleton();
        }
    }

    // crear método público y estático que me permita obtener la referencia a la clase
    public static Singleton getInstance() {
        if (instance == null) createInstance();
        return instance;
    }

    public static int getInstanciasConcurrentes(){
        return numInstancias;
    }
}
```

## ¿Como puedo probarlo?

```
public class TestSingleton {  
    public void ejecuta() {  
        for(int i = 0; i < 100; i++) {  
            new Thread(() -> Singleton.getInstance()).start();  
        }  
  
        System.out.println(Singleton.getInstanciasConcurrentes());  
    }  
  
    public static void main(String[] args) {  
        new TestSingleton().ejecuta();  
    }  
}
```



# Programación multiproceso

La programación concurrente es una forma eficaz de procesar la información al permitir que diferentes sucesos o procesos se vayan alternando en la CPU para proporcionar multiprogramación

El sistema operativo se encarga de proporcionar multiprogramación entre todos los procesos del sistema

Oculta esta complejidad tanto a los usuarios como a los desarrolladores



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

*Ejemplo: Un programador debe desarrollar un programa que lea el contenido de un fichero de texto y te cuente el número de palabras que tiene*

Si se aplica multiprogramación, el programador en vez de hacer un programa único que realice todas las funciones, realizará un programa padre que gobernará el funcionamiento de procesos hijos que realicen las funciones básicas de leer el fichero y de contar sus palabras

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

**1. Descomposición funcional**

Identificar las diferentes funciones que debe realizar la aplicación y sus relaciones

**2. Partición**

Distribuir las funciones en procesos y establecer el mecanismo de comunicación

Como son procesos cooperativos, se tendrán que comunicar y se perderá tiempo

El objetivo es maximizar la independencia entre los procesos minimizando la comunicación entre ellos

**3. Implementación**

Se realiza el desarrollo utilizando las herramientas disponibles por la plataforma

**Java** soporta la ejecución paralela de varios threads (hilos de ejecución).

Los threads en una misma máquina virtual comparten recursos, por ejemplo, memoria.

El código que ejecuta un thread se define en clases que implementan la interfaz **Runnable** o bien que extiendan de la clase **Thread**.

```
public interface Runnable {  
    public void run();  
}
```

La clase **Thread** es la clase base de Java para definir hilos de ejecución concurrentes dentro un mismo programa.

Las clases que pueden actuar de un modo concurrente deben extender de **Thread** o implementar el interfaz **Runnable**, e implementaren el método **run** el comportamiento concurrente.

La ejecución de la tarea concurrente se arranca ejecutando el método **start** si extendió de Thread, o bien si X implementó Runnable deben usar un Thread para ejecutarla (Thread t = new Thread(X); t.start(); )

## ■ Clase Thread

- Un hilo de ejecución en un programa
- Métodos
  - run() actividad del thread
  - start() activa run() y vuelve al llamante
  - join() espera por la terminación (timeout opcional)
  - interrupt() sale de un wait, sleep o join
  - isInterrupted()
  - yield()
  - stop(), suspend(), resume() (deprecated)
- Métodos estáticos
  - sleep(milisegundos)
  - currentTread()
- Métodos de la clase Object que controlan la suspensión de threads
  - wait(), wait(milisegundos), notify(), notifyAll()