
UNIDAD 1. PROGRAMACIÓN MULTIPROCESO

Autor: Joaquín Franco Ros

Versión: 15/09/2023

Este documento está bajo una licencia de Creative Commons
“Reconocimiento-NoComercial-SinObraDerivada 4.0 Internacional”.



Contenido

1. Introducción	3
2. Ejecutables procesos y servicios	3
3. Programación concurrente.....	7
3.1 Procesos concurrentes.....	8
3.2 Programación paralela y distribuida. Paso de mensajes.....	9
3.3 Ventajas de la programación concurrente	11
3.4 Hilos.	12
4. Procesos	14
4.1 Tipos de procesos	14
4.2 Estados de un proceso	14
4.3 Bloque de control de procesos	15
4.4 Cambio de contexto en la CPU.	16
4.5 Planificación de procesos.....	17
4.6 Árbol de procesos.	17
4.7 Gestión de procesos en sistemas libres y propietarios.....	18
Creación de procesos en Windows/UNIX	18
5. Programación concurrente orientada a objetos.	19
5.1 Gestión de procesos en Java.....	19
5.2 Creación de procesos en Java.	19
5.3 Sincronización entre procesos en Java.	21
5.4 Terminación de procesos en Java.	21
5.5 Comunicación de procesos en Java. Modelo de interfaz de programación para la comunicación entre procesos.....	22
5.6 Evolución de los paradigmas de comunicación entre procesos	25
6. Programación de aplicaciones multiproceso	25
6.1 Problema de los procesos lectores-escritores en un fichero compartido	25
6.2 Condiciones de competencia y regiones críticas	26
7. Documentación.	29
8. Depuración.....	29

1. Introducción

El día a día pide cada vez más potencia de cálculo a las aplicaciones informáticas, más y más cálculos numéricos en ingeniería, ciencia, astrofísica, meteorología, etc. Queremos que los ordenadores reaccionen con un razonamiento humano (inteligencia artificial). Así, tenemos ordenadores que juegan partidas de ajedrez, que hablan como humanos o que toman decisiones a partir de datos poco precisos. Aunque no es el único factor determinante, los **procesadores** juegan un papel fundamental a la hora de alcanzar esta potencia.

Cada vez son más rápidos y eficientes y los sistemas operativos permiten coordinar varios procesadores para incrementar el número de cálculos por unidad de tiempo. El uso de varios procesadores dentro de un sistema informático se llama **multiproceso**. Ahora bien, la coordinación de varios procesadores puede provocar situaciones de error que tendremos que evitar.

2. Ejecutables procesos y servicios

Es interesante distinguir entre estos conceptos:

Programas

Un **programa** es un elemento estático, un conjunto de instrucciones, unas líneas de código escritas en un lenguaje de programación, que describen el tratamiento a dar a unos datos iniciales (de entrada) para conseguir el resultado esperado (una salida concreta).

Procesos

En cambio, un **proceso** es dinámico, es una instancia de un **programa en ejecución**, que realiza los cambios indicados por el programa a los datos iniciales y obtiene una salida concreta. El proceso, además de las instrucciones, requerirá también de recursos específicos para la ejecución como el contador de instrucciones del programa, el contenido de los registros o los datos.

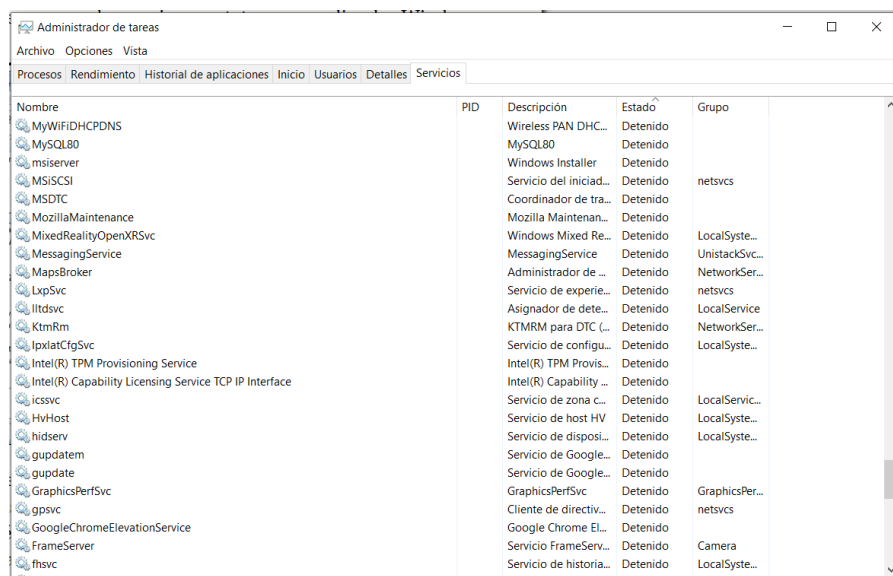
Servicios

La mayoría de nosotros tenemos en el ordenador un antivirus instalado. Cuando ponemos en marcha nuestro ordenador no arrancamos el antivirus y no interactuamos con él a menos que encuentre un virus y nos avise. Al iniciar el sistema, el programa de antivirus se ejecuta. Entonces se crea un proceso que se mantiene en ejecución hasta que apagamos nuestro ordenador. Este proceso que controla las infecciones de todos los archivos que entran en nuestro ordenador es un servicio.

Un **servicio** es un tipo de proceso que no tiene interfaz con el usuario. Puede ser inicializado por el sistema de forma automática realizando sus funciones sin que el usuario se entere. Otra opción, es que se mantenga a la espera de que alguien le haga una petición para hacer una tarea en concreto.

En Linux reciben el nombre de **demonios** y siempre acaban con la letra d como httpd el demonio del protocolo http.

En la siguiente imagen se puede ver una captura de los servicios de Windows en el *Administrador de tareas*.



Nombre	PID	Descripción	Estado	Grupo
MyWiFiDHCPDNS		Wireless PAN DHC...	Detenido	
MySQL80		MySQL80	Detenido	
msiserver		Windows Installer	Detenido	
MSISCSI		Servicio del iniciad...	Detenido	netshvc
MSDTC		Coordinador de tra...	Detenido	
MozillaMaintenance		Mozilla Maintenanc...	Detenido	
MixedRealityOpenXRSvc		Windows Mixed Re...	Detenido	LocalSyste...
MessagingService		MessagingService	Detenido	UnistackSvc...
MapsBroker		Administrador de ...	Detenido	NetworkSer...
LxpSvc		Servicio de experie...	Detenido	netshvc
lltdsvc		Asignador de dete...	Detenido	LocalService
KtmRm		KTMRM para DTC (...)	Detenido	NetworkSer...
IpxlatCfgSvc		Servicio de configu...	Detenido	LocalSyste...
Intel(R) TPM Provisioning Service		Intel(R) TPM Provis...	Detenido	
Intel(R) Capability Licensing Service TCP IP Interface		Intel(R) Capability ...	Detenido	
icsvc		Servicio de zona c...	Detenido	LocalServic...
HvHost		Servicio de host HV	Detenido	LocalSyste...
hidserv		Servicio de disposi...	Detenido	LocalSyste...
gupdate		Servicio de Google...	Detenido	
gupdate		Servicio de Google...	Detenido	
GraphicsPerfSvc		GraphicsPerfSvc	Detenido	GraphicsPer...
gpsvc		Cliente de directiv...	Detenido	netshvc
GoogleChromeElevationService		Google Chrome El...	Detenido	
FrameServer		Servicio FrameServ...	Detenido	Camera
fhsvc		Servicio de historia...	Detenido	LocalSyste...

Ejecutables

Un ejecutable es un archivo con la estructura necesaria para que el sistema operativo pueda poner en marcha el programa que hay dentro. En Windows, los ejecutables suelen ser archivos con la extensión .EXE.

Sin embargo, **Java** genera ficheros .JAR o .CLASS. Estos ficheros no son ejecutables, sino que son archivos que el intérprete de JAVA (el archivo java.exe) leerá y ejecutará.

El intérprete toma el programa y lo traduce a instrucciones del microprocesador en el que estemos, que puede ser x86 o un x64 o lo que sea. Ese proceso se hace «al instante» o JIT (Just-In-Time).



2.1 Manejando procesos en Windows

- GUI:

Presionando Ctrl+Alt+Supr y después clicando en “Administrador de tareas” en la ventana aparece el administrador de tareas.

Para más información: <https://www.howtogeek.com/405806/windows-task-manager-the-complete-guide/>

- Terminal

CMD:

- Tasklist -> muestra los procesos activos.
- Tasklist /svc -> muestra el árbol de procesos activos.
- Taskkill -> Mata a un proceso

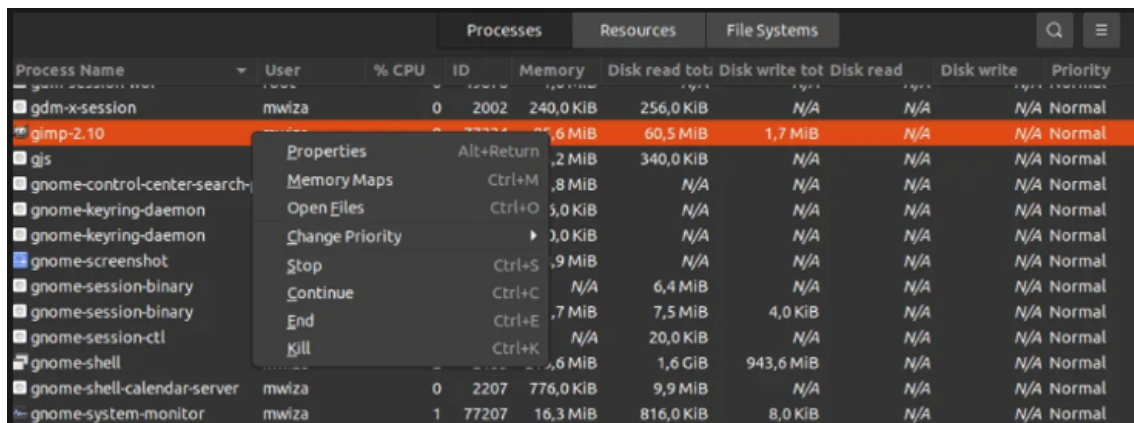
PowerShell:

- Get-Process -> muestra los procesos activos.
- Stop-Process -id -> Mata el proceso.

2.2 Manejando procesos en Linux

- GUI:

Para ver y manejar los procesos en una máquina con Linux, por ejemplo Ubuntu, se hace con el **monitor del sistema**:



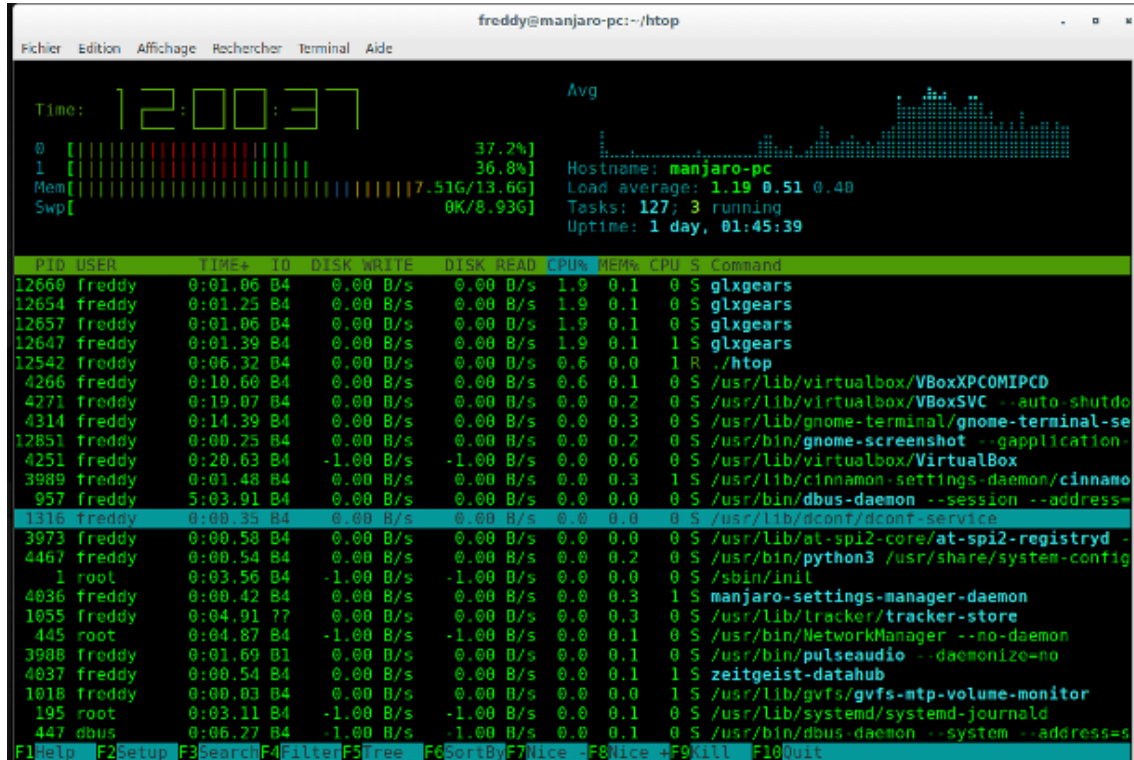
Process Name	User	% CPU	ID	Memory	Disk read tot	Disk write tot	Disk read	Disk write	Priority
gdm-x-session	mwiza	0	2002	240,0 KiB	256,0 KiB	N/A	N/A	N/A	Normal
gimp-2.10	mwiza	0	2003	25,6 MiB	60,5 MiB	1,7 MiB	N/A	N/A	Normal
gls	mwiza	0	2004	1,2 MiB	340,0 KiB	N/A	N/A	N/A	Normal
gnome-control-center-search	mwiza	0	2005	1,8 MiB	N/A	N/A	N/A	N/A	Normal
gnome-keyring-daemon	mwiza	0	2006	5,0 KiB	N/A	N/A	N/A	N/A	Normal
gnome-keyring-daemon	mwiza	0	2007	1,0 KiB	N/A	N/A	N/A	N/A	Normal
gnome-screenshot	mwiza	0	2008	1,9 MiB	N/A	N/A	N/A	N/A	Normal
gnome-session-binary	mwiza	0	2009	N/A	6,4 MiB	N/A	N/A	N/A	Normal
gnome-session-binary	mwiza	0	2010	1,7 MiB	7,5 MiB	4,0 KiB	N/A	N/A	Normal
gnome-session-ctl	mwiza	0	2011	N/A	20,0 KiB	N/A	N/A	N/A	Normal
gnome-shell	mwiza	0	2012	1,6 GiB	1,6 GiB	943,6 MiB	N/A	N/A	Normal
gnome-shell-calendar-server	mwiza	0	2207	776,0 KiB	9,9 MiB	N/A	N/A	N/A	Normal
gnome-system-monitor	mwiza	1	77207	16,3 MiB	816,0 KiB	8,0 KiB	N/A	N/A	Normal

Para más información de cómo usarlo: <https://www.makeuseof.com/manage-processes-on-ubuntu-with-system-monitor/>

- Terminal:

TOP-HTOP:

El comando *top* y el más moderno *htop* nos permite visualizar en la terminal todos los procesos con una interfaz de texto:



PS:

- ps: muestra el estado de un proceso almacenado en el directorio */proc*.
 - o Ps - -help -> muestra las opciones.
 - o Ps -> procesos ejecutándose en la sesión actual de terminal.
 - o Ps -A, ps -e -> muestra todos los procesos.
 - o Ps -u *usuario* -> muestra todos los procesos del usuario.
 - o Ps -u *usuario* | grep *aplicación* -> muestra el proceso cuya aplicación es *aplicación* y el usuario es *usuario*.
 - o Ps -eF | wc -l -> Cuenta los procesos.

PSTREE

- Pstree: muestra el árbol de procesos.
 - o Pstree -g 3 -> árbol de procesos en UNICODE.
 - o Pstree -u *usuario* -> árbol de procesos con usuarios.
 - o Pstree *usuario* -> árbol de procesos para el usuario.

JOBS

- Jobs: muestra trabajos tanto en primer plano como en segundo plano
- +. -> pone el proceso en primer plano.
- - -> pone el proceso en segundo plano.

KILL

- Kill: para un proceso.
 - Kill - -list -> muestra las opciones
 - Kill -señal PID -> manda una señal al proceso con pid PID. (-9 es cerrado forzoso).
- Killall PID -> mata todos los procesos.

Actividad 1:**Prueba 1:**

En los sistemas operativos Windows podemos usar desde la línea de comandos la orden **tasklist** para ver los procesos que se están ejecutando. Ejecuta la orden e investiga los procesos actuales y el significado de las columnas que aparecen.

Prueba 2:

El administrador de tareas en Microsoft Windows[CTRL+Alt+Supr] permite gestionar los procesos del sistema operativo. En él se puede ver qué procesos se están ejecutando con su uso de procesador y memoria.

- Abre el administrador de tareas y comprueba para tres procesos cuánta memoria y CPU están usando.

- Abre la calculadora de Windows y termina su proceso con el administrador.

Pruebas en Linux: Prueba los comando vistos en este epígrafe en el sistema operativo Ubuntu para que te familiarices con ellos.

3. Programación concurrente

Para explicar de una manera sencilla la **conurrencia** vamos a hacer una analogía con la elaboración de un plato de cocina. Un cocinero que tenga intención de hacer un plato muy 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.

El cocinero 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.

Gracias a la evolución que ha sufrido el hardware durante los últimos años se han creado sistemas operativos que pueden optimizar los recursos de los procesadores y pueden ejecutar diferentes procesos de forma simultánea. La ejecución simultánea de procesos se denomina **conurrencia**.

3.1 Procesos concurrentes

Actividad 2: Analogía del cocinero con procesos concurrentes

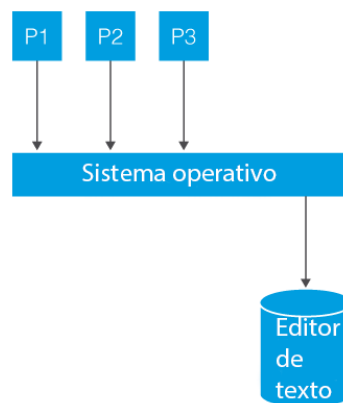
Siguiendo con la analogía del cocinero asocia los siguientes términos:

Proceso, procesador, resultados y programa.

El **sistema operativo** es el encargado de la gestión de procesos. Los crea, los elimina y los provee de instrumentos que permitan 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) con sus propias variables, pilas y registros.

Imaginemos un servidor de aplicaciones en el que tenemos instalado un programa de edición de texto. Supongamos que existen varios usuarios que quieren escribir sus textos ejecutando el editor. Cada instancia del programa es un proceso totalmente independiente a los demás. Cada proceso tiene unos datos de entrada y por lo tanto diferentes salidas. Cada usuario escribirá en su ejecución del editor (el proceso), su texto.

En la siguiente figura se refleja este ejemplo:



Cuando un proceso se encuentra en ejecución se encuentra completamente en memoria y tiene asignados los recursos que necesita. Un proceso no puede escribir en zonas de memoria asignada a otros procesos, la memoria no es compartida.

La programación concurrente:

- Da mecanismos para la comunicación y sincronización entre procesos que se ejecutan de forma simultánea en un Sistema Informático.
- Permite definir qué instrucciones de nuestros procesos se pueden ejecutar de forma simultánea con los otros procesos sin que haya errores; y cuáles tienen que ser sincronizadas con las de otros procesos para obtener resultados correctos.

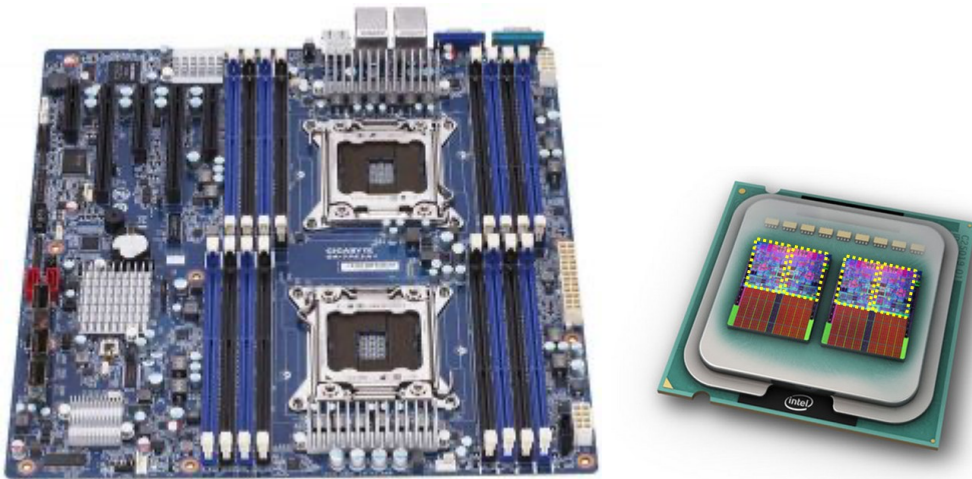
3.2 Programación paralela y distribuida. Paso de mensajes.

En este apartado vamos a ver tres tipos de programación concurrente: multiprogramación, programación paralela o multiproceso y programación distribuida.

Hasta ahora hemos hablado de elementos de software, tales como programas y procesos, pero no del hardware utilizado para ejecutarlos. El elemento de hardware en el que se ejecutan los procesos es el **procesador**. Un procesador es el componente de hardware de un sistema informático encargado de ejecutar las instrucciones y procesar los datos. Cuando un dispositivo informático está formado por un único procesador hablaremos de **sistemas monoprocesador**, por el contrario, si está formado por más de un procesador hablaremos de **sistemas multiprocesador**.

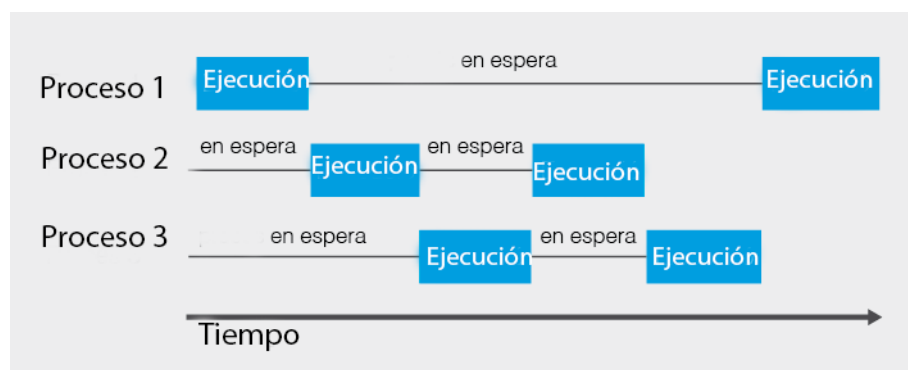
Un **sistema monoprocesador** es aquel que está formado únicamente por un procesador.

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

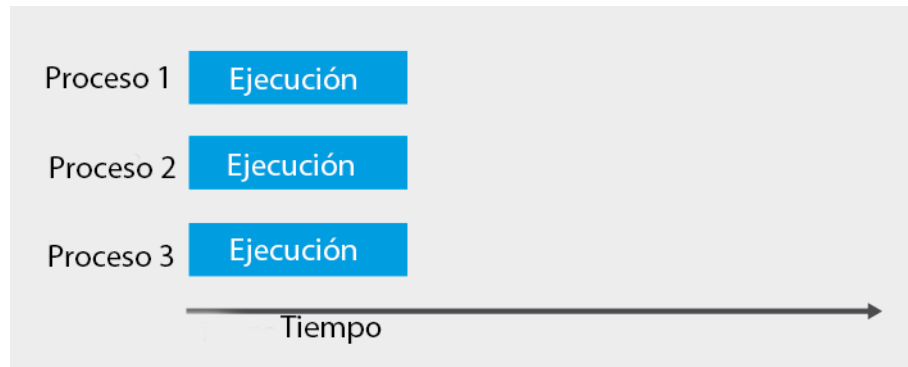


Placa base multiprocesador

En un **sistema informático monoprocesador** se va alternando la ejecución de los procesos, es decir, se quita un proceso de la CPU, se ejecuta otro y se vuelve a colocar el primero sin que se entere de nada; esta operación se realiza tan rápido que parece que cada proceso tiene dedicación exclusiva. En este caso hablamos de **multiprogramación**.



En un **sistema informático multiprocesador** existen dos o más procesadores, por lo tanto, se pueden ejecutar simultáneamente varios procesos. También se pueden calificar de multiprocesadores aquellos dispositivos donde el procesador tiene más de un **núcleo (multicore)**, es decir, tienen más de una CPU en el mismo circuito integrado del procesador. En este caso hablamos de **programación paralela o multiproceso**.



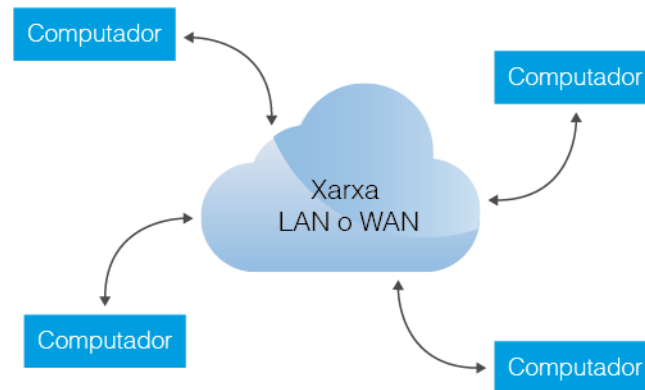
Ejecución paralela en tres procesadores 1

En un dispositivo multiprocesador la concurrencia es real, los procesos son ejecutados de forma simultánea en diferentes procesadores del sistema. 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.

El sistema operativo Windows 95 únicamente permite trabajar con un único procesador. Los sistemas operativos a partir de Win2000 / NT y Linux / Unix son multiproceso, pueden trabajar con más de un procesador.

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. Un ejemplo de sistema distribuido puede ser el de un banco con muchas oficinas en el mundo, con un ordenador central para oficina para guardar las cuentas locales y hacer las transacciones locales. Este equipo puede comunicarse con los otros ordenadores centrales de la red de oficinas. Cuando se hace una transacción no importa donde se encuentra la cuenta o el cliente.

La comunicación entre procesos para intercambiar datos o sincronizarse entre ellos se hace con **mensajes** que se envían a través de la red de comunicaciones que comparten.



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.

La programación MPI es una propuesta de estándar para un interfaz de **paso de mensajes** en entornos paralelos, especialmente con memoria distribuida. En este modelo, una ejecución consta de uno o más procesos que se comunican llamando a rutinas de una biblioteca para recibir y enviar mensajes entre procesos.

3.3 Ventajas de la programación concurrente

La concurrencia tiene los siguientes beneficios:

1. Mejorar el aprovechamiento de la CPU
2. Velocidad de ejecución. AL repartir un programa en subprocesos, estos se pueden repartir entre procesadores o gestionarse en uno solo según su importancia.
3. Soluciona problemas de naturaleza concurrente:
 - **Sistemas de control:** normalmente captura de datos por sensores, análisis y actuación en función de ellos. Ejemplo: sistemas de tiempo real
 - **Tecnología web:** Los servidores web atienden peticiones simultáneas concurrentemente.
 - **Aplicaciones basadas en GUI:** El usuario interactúa con la aplicación mientras la aplicación hace alguna tarea. Ejemplo: navegador descargando mientras estamos navegando.
 - **Simulación:** programas que modelan sistemas físicos con autonomía
 - **Sistemas Gestores de Bases de Datos:** Cada usuario que interactúa con el gestor puede tratarse como un proceso.

Existen tres tipos básicos de interacción entre procesos concurrentes:

- **Independientes.** Sólo interfieren en el uso de la CPU
- **Cooperantes.** Un proceso genera la información o servicio que otro necesita
- **Competidores.** Procesos que necesitan usar los mismos recursos de forma exclusiva.

3.4 Hilos.

Recordemos el cocinero que trabaja simultáneamente preparando un plato para diferentes comensales. El procesador de un sistema informático (cocinero) está ejecutando diferentes instancias del mismo programa (la receta). Aparte, el cocinero haciendo diferentes partes del plato. Si la receta es morcilla con patatas, dividirá su tarea en preparar la morcilla por un lado y las patatas por la otra. Ha dividido el proceso en dos subprocesos. Cada uno de estos subprocesos denominan **hilos**.

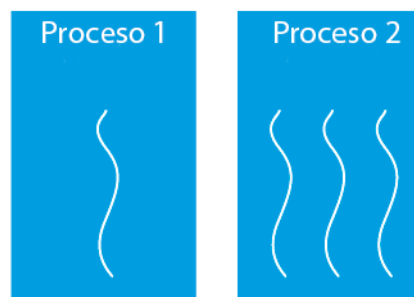
El sistema operativo puede mantener en ejecución varios procesos a la vez utilizando concurrencia o paralelismo. Además, **dentro de cada proceso pueden ejecutarse varios hilos**, por lo que bloques de instrucciones que presenten cierta independencia puedan ejecutarse a la vez dentro del mismo proceso.

Los hilos comparten los recursos del proceso (datos, código, memoria, etc). En consecuencia, los hilos, a diferencia de los procesos, comparten memoria y si un hilo modifica una variable del proceso, el resto de hilos podrán ver la modificación cuando accedan a la variable.

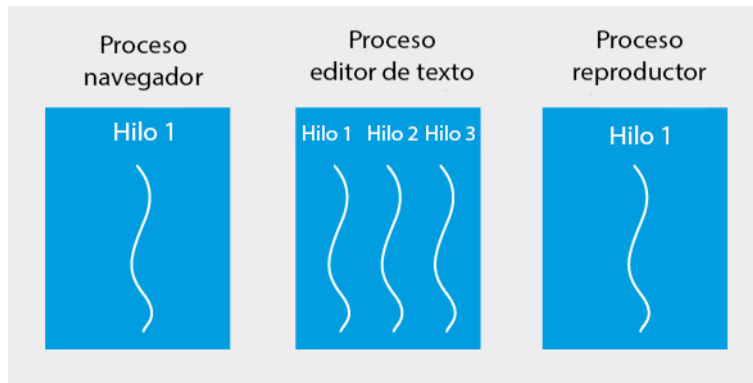
Los procesos son llamados entidades pesadas porque están en espacios de direccionamiento de memoria independientes, de creación y de comunicación entre procesos, lo que consume muchos recursos de procesador. En cambio, ni la creación de hilos ni la comunicación consumen mucho tiempo de procesador. De ahí que los hilos se denominan **entidades ligeras**.

Un proceso estará en ejecución mientras alguno de sus hilos esté activo. Sin embargo, también es cierto que, si finalizamos un proceso de forma forzada, sus hilos también terminarán la ejecución.

En la siguiente figura se puede ver un proceso con un hilo y proceso con tres hilos:



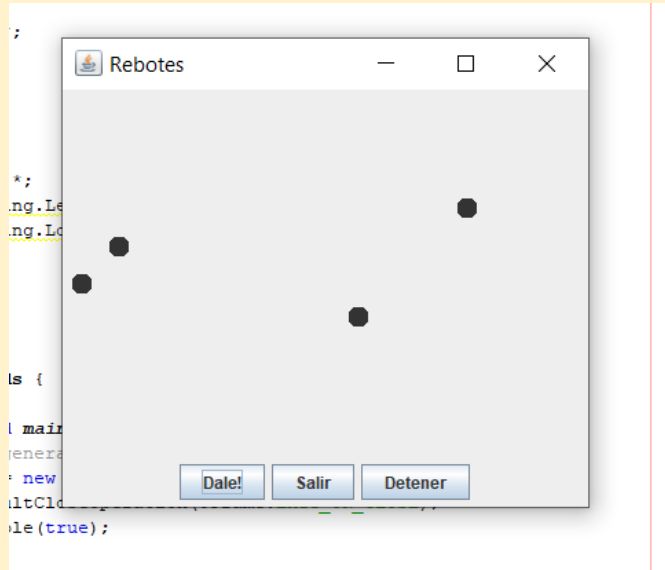
Los sistemas operativos actuales admiten la concurrencia, es decir, pueden ejecutar diferentes procesos a la vez. Podemos estar navegando por Internet con un navegador, escuchando música con un reproductor y, con un editor de texto, redactando un documento. Cada aplicación que se ejecuta es un proceso y cada uno de estos procesos tiene como mínimo un hilo de ejecución, pero podrían tener más de uno. El editor de texto podría tener un hilo que se encargara de ver qué letra estamos escribiendo, otro que vaya comprobando la ortografía y otro que de vez en cuando se active para guardar el documento. Hay que ver que un hilo pertenece a un proceso y que los diferentes hilos pueden hacer tareas diferentes dentro de su proceso.



Algunos ejemplos de utilización de hilos podrían ser:

- Pestañas abiertas del navegador Google Chrome
- Word tiene un hilo comprobando automáticamente la gramática a la vez que está escribiendo un documento.
- Diferentes personajes de un videojuego moviéndose e interactuando independientemente.
- Etc.

Actividad 3. Programa crea bolas que rebotan en la pantalla. Cada bola es un hilo de ejecución dentro del mismo proceso.



En este programa, cada vez que se le da al botón **Dale!**, se crea un hilo nuevo que ejecuta el código del rebote de la bola dentro del mismo proceso.

La bola no es más que un objeto con dos coordenadas x e y que van cambiando dentro de un bucle while para representar su movimiento.

4. Procesos

Como ya se definió, un proceso es cualquier **programa en ejecución**. Un proceso necesita ciertos recursos para realizar satisfactoriamente su tarea. Estos son principalmente:

- Tiempo de CPU
- Memoria
- Archivos
- Dispositivos de entrada-salida E/S.

Las obligaciones del **sistema operativo** como gestor de procesos son:

- La creación y eliminación de los procesos
- La planificación
- Establecimiento de mecanismos para la sincronización y comunicación de procesos.

Un proceso está formado por:

- Sección de texto (código del programa).
- Actividad actual representada por: el valor de contador del programa (CP) y el contenido de los registros del procesador.
- Pila que contiene los datos temporales (parámetros de subrutinas, direcciones de retorno y variables locales).
- Sección de datos que contiene las variables globales y la memoria utilizada por el programa.

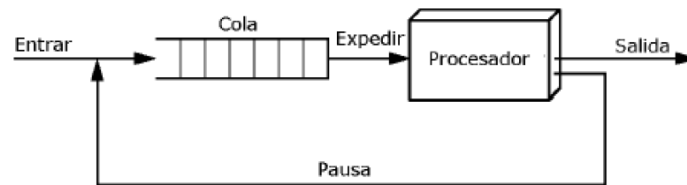
4.1 Tipos de procesos

Los procesos se pueden clasificar en:

- **Por lotes:** Están formados por una serie de tareas, de las que el usuario sólo está interesado en el resultado final (imprimir documentos).
- **Interactivos:** Aquellas tareas en las que el proceso interactúa continuamente con el usuario y actúa de acuerdo a las acciones que éste realiza, o a los datos que suministra (procesador de textos).
- **Tiempo real:** Tareas en las que es crítico el tiempo de respuesta del sistema (programa que controla un brazo mecánico).

4.2 Estados de un proceso

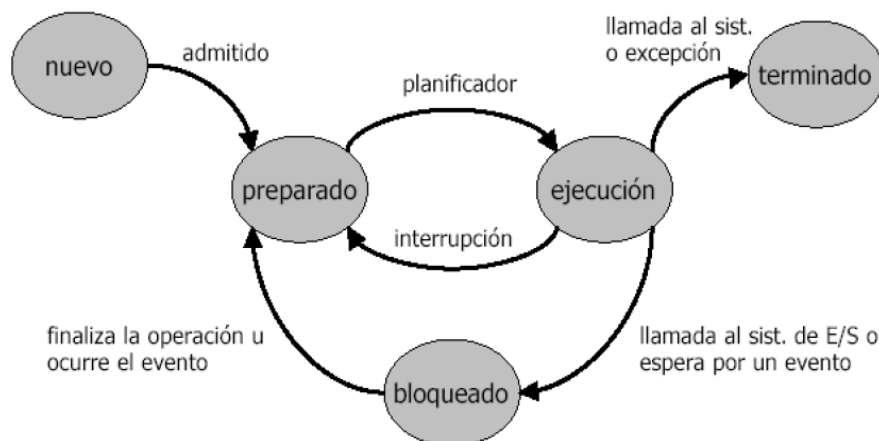
Todos los sistemas operativos disponen de un **planificador de procesos** encargado de repartir el uso del procesador de la forma más eficiente posible y asegurando que todos los procesos se ejecuten en algún momento. Para realizar la planificación, el sistema operativo se basará en el estado de los procesos para saber qué necesitarán el uso del procesador. Los procesos en disposición de ser ejecutados organizarán en una cola esperando su turno.



A medida que un proceso se ejecuta cambia de estado. Cada proceso puede estar en uno de los estados:

- **Nuevo (new):** el proceso se está creando.
- **En Ejecución:** el proceso está en la CPU ejecutando instrucciones.
- **Bloqueado:** el proceso está esperando a que ocurra un suceso (ej. terminación de E/S o recepción de una señal).
- **Listo:** Esperando a que se le asigne un procesador.
- **Terminado:** finalizó su ejecución, por tanto, no ejecuta más instrucciones y el SO le retirará los recursos que consume.

Nota: Sólo un proceso puede estar ejecutándose en cualquier procesador en un instante dado, pero muchos procesos pueden estar listos y esperando.



4.3 Bloque de control de procesos

Cada proceso tiene una estructura de datos en la que se guarda la información asociada a la ejecución del proceso. Esta zona se denomina **Bloque de Control de Proceso (BCP)**.

Puntero	Estado
Identificador de proceso	
Contador de programa	
Registros	
Límites de memoria	
Estado de la E/S ...	

La información guardada en el PCB es la siguiente:

- **Identificador del proceso o PID:** Es un número único para cada proceso, como un DNI de proceso.
- **Estado** actual del proceso: en ejecución, listo, bloqueado, suspendido, finalizando.
- **Espacio de direcciones de memoria** donde comienza la zona de memoria reservada al proceso y su tamaño.
- **Información para la planificación:** prioridad, quantum, estadísticas, ...
- **Información para el cambio de contexto:** valor de los registros de la CPU, entre ellos el contador de programa y el puntero a pila. Esta información es necesaria para poder cambiar de la ejecución de un proceso a otro.
- **Recursos utilizados.** Ficheros abiertos, conexiones, ...
- **Registros:** se guarda la información que en cada instante necesita la instrucción que está ejecutando la CPU.
 - Destacamos dos:
 - **Contador del programa:** dirección de la siguiente instrucción a ejecutar.
 - **Puntero a la pila:** apunta a un trozo de memoria donde se guarda el contexto de la CPU.

4.4 Cambio de contexto en la CPU.

El **cambio de contexto** consiste en **desalojar a un proceso de la CPU y reanudar otro**. Se guarda el estado del proceso saliente en su PCB y se recuperan los registros del proceso que entra.

El cambio de contexto (tiempo de conmutación) **es tiempo perdido y debe ser lo más rápido posible**.

El hardware en ocasiones facilita el cambio de contexto, haciendo que un cambio simplemente implique cambiar el puntero al conjunto de registros actual.

4.5 Planificación de procesos.

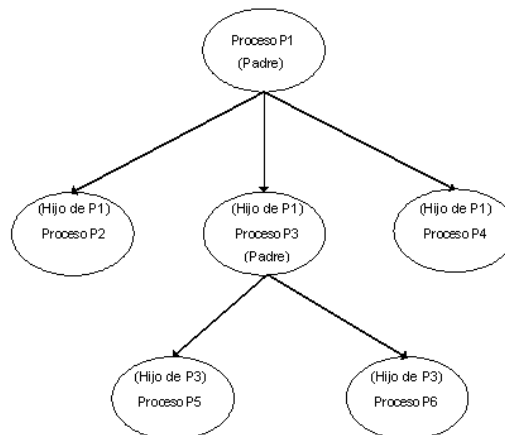
El sistema operativo tiene un proceso llamado planificador que se encarga de decidir qué proceso se ejecuta y durante cuánto tiempo. La decisión la toma en base a un algoritmo de planificación, algunos de ellos son:

- **Round robin:** Cada proceso se ejecuta durante un quantum. Si no le ha dado tiempo a finalizar durante ese quantum se coloca al final de la cola de procesos listos y espera de nuevo su turno.
- **Por prioridad:** Se asignan prioridades a los distintos procesos. Se ejecutan antes los que tienen mayor prioridad.
- **Múltiples colas:** Combinación de los dos anteriores. Todos los procesos con una misma prioridad están en una misma cola y cada cola se gestiona con Round robin.

4.6 Árbol de procesos.

Un nuevo proceso se crea siempre por petición de otro proceso. Cualquier proceso en ejecución depende del proceso que lo creó estableciéndose un vínculo entre ambos.

Cuando se arranca el ordenador y se carga el *kernel* del sistema en disco se crea el proceso inicial del sistema. A partir de este proceso se crea el resto de procesos de manera jerárquica, creando lo que se denomina el **árbol de procesos**. En la siguiente figura se pueden ver los procesos padres e hijos:



4.7 Gestión de procesos en sistemas libres y propietarios

Las operaciones básicas en cualquier sistema operativo son:

- **Create:** Creación de un nuevo proceso.

Cuando se crea un proceso, padre e hijo se ejecutan concurrentemente, ambos comparten la CPU y se irán intercambiando según la política de planificación.

- **Wait:** Espera de un proceso.

Si el proceso padre necesita esperar hasta que el hijo termine su ejecución para poder continuar la suya con los resultados obtenidos del hijo.

- **Exit:** Terminación de un proceso

Se liberan los recursos utilizados por el proceso.

Dependiendo del SO se produce “terminación en cascada”, es decir, si el padre termina tienen que terminar también todos sus hijos.

- **Destroy:** Terminación abrupta de un proceso hijo por el padre.

Creación de procesos en Windows/UNIX

La creación de procesos en los sistemas operativos se realiza mediante las **llamadas al sistema**.

Cada SO operativo gestiona sus procesos de manera diferente.

- **Windows:** función ***createProcess()*** que crea un nuevo proceso hijo a partir de un programa distinto al que está en ejecución.
- **UNIX:** función ***fork()***, que crea un proceso hijo con un duplicado del espacio de direcciones del padre, es decir, un duplicado del programa que se ejecuta desde la misma posición.

Cada proceso es independiente y tiene su propio espacio de memoria asignado (imagen de memoria).

Los procesos pueden compartir recursos (ficheros y memoria compartida) para intercambiarse información.

5. Programación concurrente orientada a objetos.

5.1 Gestión de procesos en Java.

Cada sistema operativo tiene características únicas y la gestión de procesos es diferente. Java evita tener que depender del Sistema Operativo para la gestión de procesos. La JVM permite la ejecución de binarios de Java sobre cualquier sistema operativo. *Write Once, Run Anywhere* “escribe una vez y ejecuta en cualquier lugar”.

5.2 Creación de procesos en Java.

En java la clase que representa a un proceso se llama **Process**. Existen dos maneras de crear objetos de tipo **Process**: utilizando la clase **ProcessBuilder** o utilizando la clase **Runtime**.

Actividad 4: ejemplo con la clase **Runtime** donde se lanza el proceso Notepad:

```
8 import java.io.IOException;
9
10 public class Ejemplo1 {
11     public static void main(String[] args) {
12         Runtime rt = Runtime.getRuntime(); //Objeto runtime asociado a la aplicacion
13         String [] comando = {"notepad"}; // comando
14         Process p; //Para el objeto proceso
15
16         try {
17             p= rt.exec(comando);
18         } catch (IOException e){
19             System.out.println ("Error al ejecutar el comando: " + comando);
20             e.printStackTrace(); //para que me printe error estandar
21         }
22     }
23 }
24
25
```

Como se puede observar dos métodos muy importantes de esta clase son:

- *Static Runtime getRuntime()*: devuelve el objeto **Runtime** asociado con la aplicación Java en curso.
- *Process exec(String []comando+argumentos)*: ejecuta la orden especificada en *comando* en un proceso separado. Devuelve un objeto **Process**, que se puede utilizar para controlar la interacción del programa Java con el nuevo proceso en ejecución. La orden puede ser cualquier comando del sistema operativo. Puede lanzar varias excepciones como: *SecurityException* (si existe un gestor de seguridad y no se permite la ejecución), *IOException* (error de E/S), *NullPointerException* (el comando es nulo) y *IllegalArgumentException* (argumento ilegal en el comando).

Documentación sobre la clase **Process**:

<https://docs.oracle.com/en/java/javase/18/docs/api/java.base/java/lang/Process.html>

Documentación sobre la clase **Runtime**:

<https://docs.oracle.com/en/java/javase/18/docs/api/java.base/java/lang/Runtime.html>

Actividad 4.2: el mismo ejemplo con la clase **ProcessBuilder**.

```
15 public class Ejemplo12 {
16
17     public static void main(String[] args) {
18
19         String [] comando = {"notepad"}; // comando
20         ProcessBuilder pb = new ProcessBuilder(comando); //Objeto ProcessBuilder asociado a la aplicacion
21         Process p; //Para el objeto proceso
22
23         try {
24
25             //bp.command(comando); //Otra forma de asignar el comando en PB
26             p=pb.start();
27         }catch (IOException e){
28             System.out.println ("Error al ejecutar el comando: " + comando);
29             e.printStackTrace(); //para que me printe error estandar
30         }
31     }
32 }
33
34
```

En esta clase permite crear procesos indicando un conjunto de opciones en la creación de los mismos, los métodos más importantes son:

- En este caso primero se obtiene un objeto de tipo **ProcessBuilder** con su constructor al que podemos pasar el comando a ejecutar. *ProcessBuilder builder = new ProcessBuilder (String [] comando+argumentos).*
- *Processbuilder= builder.start()* donde se inicia el nuevo proceso con el comando especificado previamente.

Hay que tener en cuenta que un objeto de tipo **ProcessBuilder** tiene otros métodos que permiten definir opciones del proceso antes de su creación, como argumentos, el entorno, directorio de trabajo, entrada estándar, salida y salida de error estándar, interacción con ficheros, etc. Es decir, nos ofrece más control sobre los procesos que **Runtime**.

Documentación sobre la clase **ProcessBuilder**:

<https://docs.oracle.com/en/java/javase/18/docs/api/java.base/java/lang/Runtime.html>

5.3 Sincronización entre procesos en Java.

En Java, el proceso padre puede esperar a que termine el hijo y obtener su información de finalización. La método `waitFor()` de la clase **Process** bloquea al padre hasta que el hijo finaliza mediante un `exit`.

En el trozo de código siguiente se puede observar cómo se obtiene este valor de la aplicación del ejemplo anterior. Se puede forzar a que el proceso termine mal poniendo como comando uno inexistente como **VERR**.

```
//compruebo error de terminacion del proceso
try{
    exitValue = p.waitFor();
    System.out.println("El proceso termina (0 = bien, 1 = mal): " + exitValue);
}catch (InterruptedException ex){
    ex.printStackTrace();
}
```

También es interesante el método `exitValue()` de la clase **Process** que retorna el valor de retorno del proceso hijo. El proceso ha de haber terminado antes de obtener el valor. Si no, se lanzará la excepción **IllegalThreadStateException**.

5.4 Terminación de procesos en Java.

Un proceso puede terminar de dos maneras en un sistema operativo:

- **Terminación normal:** donde el proceso hijo realiza su ejecución completa y termina cuando ejecuta la operación `exit`.
- **Terminación abrupta:** donde un proceso puede terminar de forma abrupta un proceso hijo que creó utilizando la operación `destroy` liberando sus recursos en el sistema operativo subyacente.

Actividad 5: Realiza la hoja de ejercicios 1. Lanzar comandos en Java

Para los comandos de Windows que no tienen ejecutable directo (como por ejemplo DIR o ATTRIB) es necesario utilizar el comando CMD.exe. Entonces para hacer un DIR desde un programa Java tendríamos que escribir en comando lo siguiente: "CMD /C DIR". El argumento **/C** ejecuta el comando especificado y luego lo finaliza.

Como se puede ver en este ejemplo donde se intenta lanzar el comando **VER**, aquí se ha creado un array de String donde en cada posición está primero el comando y luego sus argumentos `/c` y luego **VER**.

```
String []comando = {"CMD", "/c", "VER"};
```

Actividad 6: Intenta crear un programa con *Runtime* que lance el comando VER.

¿Obtienes alguna salida?

Si no se obtiene una salida es debido a que la salida del comando se redirige a nuestro programa Java, no a la pantalla, es decir, lo que nos devuelve el método *exec* del **Runtime**, tenemos que usar el objeto **Process**. La clase *Process* posee el método *getInputStream()* que nos permite leer el stream de salida del proceso, es decir, podemos leer lo que el comando que ejecutamos escribió en la consola.

Actividad 7: Lanza el comando *IPCONFIG /ALL* y se guarda la salida en un fichero.

5.5 Comunicación de procesos en Java. Modelo de interfaz de programación para la comunicación entre procesos.

En general la comunicación consiste en que un proceso da o deja información y luego otro proceso recibe y recoge información.

Los lenguajes de programación y los SO proporcionan primitivas de sincronización que facilitan la interacción entre procesos.

Las interacciones entre procesos y con el resto del sistema (recursos, otros procesos); podemos clasificarlas en:

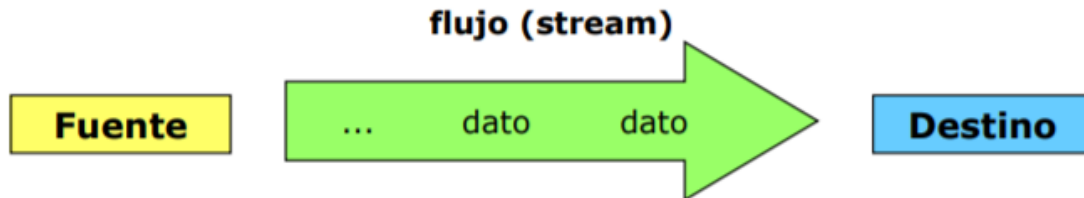
- 1) **Sincronización:** un proceso puede conocer el punto de ejecución en el que está otro proceso en un instante determinado
- 2) **Exclusión mutua:** mientras un proceso accede a un recurso, ningún otro puede acceder al mismo recurso o variable compartida
- 3) **Sincronización condicional:** Solo se accede a un recurso cuando se encuentra en un determinado estado interno.

En **Java** usamos sockets y buffers como cualquier otro *stream* o flujo de datos. Usaremos los métodos **read/write** en lugar de *send/receive*.

Las lecturas/escrituras, serán bloqueantes. Un proceso quedará bloqueado hasta que los datos estén listos para ser leídos. En escritura bloqueará al proceso que intenta escribir hasta que el recurso no esté preparado para poder escribir.

Entendiendo los streams en Java

En Java se define la abstracción de **stream** (flujo) para tratar la comunicación de información entre una fuente y un destino donde fluye una secuencia de datos. La fuente y el destino pueden ser dispositivos de distintas clases (ficheros, teclado, red, procesos, ...).



Además, la comunicación se puede hacer de distintas formas. Por ejemplo, con modo de acceso secuencial, aleatorio o que la información intercambiada esté formato en binario, caracteres, o líneas completas.

Es importante saber que todo proceso tiene una serie de flujos estándar:

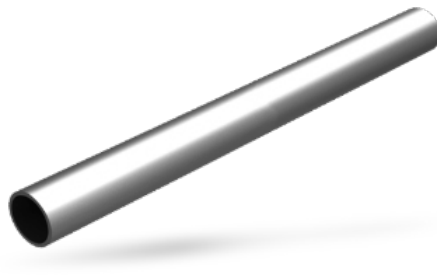
- **Una entrada estándar (stdin):** habitualmente el teclado.
- **Una salida estándar (stdout):** habitualmente la consola.
- **Una salida de error estándar (stderr):** habitualmente la consola.

En Java estos flujos están imp por los objetos:

- **System.in:** es una instancia de **InputStream** que representa un flujo de bytes de entrada.
 - El método más importante es *read()* que permite leer un byte de la entrada como entero.
- **System.out:** que es una instancia de **PrintStream** (subclase de **OutputStream**) que representa un flujo de bytes de salida y permite formatearla.
 - Los métodos más importantes son *print()* y *println()* o *flush()* que vacía el buffer de salida escribiendo su contenido.
- **System.err:** funcionamiento similar a **System.out** y se utiliza para enviar mensajes de error.



La comunicación se puede ver como un tubo (pipe) por donde va el flujo de información que se intercambian los dos dispositivos que están conectados a sus extremos.



En nuestro caso los dispositivos de entrada y salida son **dos procesos** que se quieren comunicar. Para ello debemos hacer uso de los siguientes métodos de la clase **Process**:

- `getOutputStream()`: se obtiene el flujo de salida (OutputStream) conectado a la entrada estándar del subprocesso. Este flujo servirá normalmente para que el proceso padre le mande información al proceso hijo.
- `getInputStream()`: se obtiene el flujo de entrada (InputStream) conectado a la salida estándar del subprocesso. Este flujo servirá normalmente para que el proceso padre obtenga información del proceso hijo.
- `getErrorStream()`: devuelve el flujo de salida conectado a la entrada normal del subprocesso. En Java hay que saber que `stderr` y `stdout` están conectados ambos al mismo dispositivo (consola).
*Nota: se puede utilizar `redirectErrorStream(boolean)` de la clase **ProcessBuilder** para separarlos.*

Para leer la salida de un proceso nosotros necesitamos las siguientes clases:

- `InputStream` -> lee del stream de bytes.
- `InputStreamReader` -> transforma los bytes a caracteres.
- `BufferedReader` -> almacena en un buffer el stream de caracteres para poder leerlos de manera más eficiente y fácil.

Sintaxis

- 1) Obteniendo el flujo de entrada del proceso hijo:

```
BufferedReader br =
    new BufferedReader (
        new InputStreamReader (in: p.getInputStream()));
```

- 2) Leyendo la salida del hijo línea a línea:

```
while ((line=br.readLine())!=null){
    System.out.println (x: line);
}
```

- 3) Cerrando el stream

Existen dos opciones:

- Cerrando manualmente con el método `.close()`.
- Cerrando automáticamente usando *“try-catch with resources”*.

Actividad 8: Añade la actividad 6 el código necesario para mostrar la salida de error estándar por consola cuando se ejecute el comando VERR (comando que no existe).

5.6 Evolución de los paradigmas de comunicación entre procesos

Los Mecanismos para intercambiar información han sido de dos tipos:

- Intercambio de mensajes: Primitivas send, recibe o wait
- Recursos o memoria compartidos: Primitivas write o read para escribir y leer recursos.

La comunicación entre procesos dentro del mismo sistema se puede hacer con intercambio de mensajes:

- Usando un buffer de memoria
- Usando un socket

El **socket** se utiliza para intercambiar información entre procesos en distintas máquinas a través de la red (los utilizaremos en el tema 3).

El **buffer de memoria** crea un canal de comunicación entre dos procesos utilizando la memoria principal del sistema.

6. Programación de aplicaciones multiproceso

6.1 Problema de los procesos lectores-escriptores en un fichero compartido

En el caso de lecturas y escrituras en un fichero, tenemos que diferenciar si solo leemos, solo escribimos o hacemos ambas cosas a la vez, ya que, según el caso puede ser necesario sincronizar dichos procesos.

- Si el acceso es de solo lectura no hay problema, todos los procesos pueden acceder simultáneamente para leer del mismo fichero
- Si el acceso es de escritura o de lectura/escritura, el SO permite pedir un acceso de forma exclusiva al fichero. Eso implica que el proceso tiene que esperar a que todos los lectores terminen. Cuando le llega el turno al escritor, los procesos lectores esperarán a que termine.

Actividad Acceso a un recurso compartido sin mecanismo de sincronización.

Escribimos dos programas: FicheroAccesoMultiple y MultiplesAccesoFichero

Programa 1: Aplicación que accede a un fichero que contiene un valor entero, lee el valor, lo incrementa en 1 y escribe el valor actualizado en el mismo fichero, algo así:

```
Valor= LeerValorFichero (fichero);  
Valor ++;
```

EscribirValorFichero (fichero, Valor);

Gestionar las excepciones que se pueden producir para los accesos al fichero.

Lógica:

- Comprobar si pasamos fichero donde redirigir la salida (javalog.txt) por la línea de argumentos, en caso contrario intentar crearlo y sin avisar de que no podemos redirigir e irnos.
- Tenemos que redirigir el flujo de salida estándar al fichero javalog.txt (System.setOut (PrintStream))
- Preparamos el fichero donde vamos a leer y escribir el número, por ejemplo, "accesomultiple.txt". Si el fichero no existe, lo creamos.
-

Programa 2: Aplicación que crea **20 procesos** de la anterior. De forma que todos los procesos utilizarán el mismo fichero para realizar las instrucciones.

Nota:

- Retocar Programa 1 para que el número de orden del proceso y el archivo se pasen por la línea de argumentos, porque vamos a ejecutar el Programa 2 invocando 20 veces al programa 1, en la forma:
`p = rt.exec("java -jar " + "FicheroAccesoMultiple.jar " + i + " nuevo.txt");`
- Para probar: Poner en el mismo directorio el .jar del Programa1 y del Programa 2

Vemos que la lectura-incremento-escritura del valor, se debe ejecutar como una unidad y de forma exclusiva, para evitar que se mezclen las lecturas de unos y otros procesos, que es la principal causa de que no podamos determinar el contenido final del fichero. **¿Qué está pasando?**

Otro ejemplo parecido:

Cuando no hay control sobre una sección crítica se puede ilustrar con el ejemplo de dos procesos que quieren modificar una variable común x. El proceso A quiere incrementarla: x++. El proceso B disminuirla: x-- ;. Si los dos procesos acceden a leer el contenido de la variable a la vez, los dos obtendrán el mismo valor, si hacen su operación y guardan el resultado, este será inesperado. Dependerá de quien salve el valor de x en último lugar.

6.2 Condiciones de competencia y regiones críticas

Región crítica = **conjunto de instrucciones en las que un proceso accede a un recurso compartido.**

Las instrucciones que forman la región crítica se tienen que ejecutar de forma indivisible o atómica y de forma exclusiva con respecto al resto de procesos que accedan al mismo recurso compartido.

Al identificar y definir nuestras regiones críticas en el código, tendremos en cuenta:

- **Se protegerán con secciones críticas sólo aquellas instrucciones que acceden a un recurso compartido.**

- Las **instrucciones que forman una sección crítica serán las mínimas**. Incluirán sólo las instrucciones imprescindibles que deban ser ejecutadas de forma atómica.
- Se pueden **definir tantas secciones críticas como sean necesarias**.
- **Un único proceso entra en su sección crítica**. El resto de procesos esperan a que éste salga de su sección crítica. El resto de procesos esperan, porque encontrarán el **recurso bloqueado**. El proceso que está en su sección crítica, es el que ha bloqueado el recurso.
- Al **final de cada sección crítica**, el **recurso** debe ser **liberado** para que puedan utilizarlo otros procesos.

Algunos lenguajes de programación permiten definir bloques de código como secciones críticas. Estos lenguajes, cuentan con palabras reservadas específicas para la definición de estas regiones.

```
Entrada_Sección_Critica /* Solicitud per ejecutar Sección
Crítica */

/* código Sección Crítica */

Salida_Sección_Critica /* otro proceso puede ejecutar la Sección
Crítica */
```

La **Entrada_Sección_Critica** representa la parte del código en la que los procesos piden permiso para entrar en la sección crítica. La **Salida_Sección_Critica** en cambio, representa la parte que ejecutan los procesos cuando salen de la sección crítica liberando la sección y permite a otros hilos entrar.

*** En Java, veremos cómo definir este tipo de regiones a nivel de hilo en posteriores unidades.**

****Hay que tener mucho cuidado con los mecanismos de sincronización se pueden producir condiciones no deseables que hagan que el programa se quede colgado o no funcione correctamente. En el siguiente tema veremos qué condiciones se pueden dar y cómo resolverlas.**

A nivel de procesos, lo primero, haremos, que nuestro ejemplo de accesos múltiples a un fichero sea correcto para su ejecución en un entorno concurrente. En esta presentación identificaremos la sección o secciones críticas y qué objetos debemos utilizar para conseguir que esas secciones se ejecuten de forma excluyente.

En nuestro ejemplo, vamos a ver cómo definir una sección crítica para proteger las actualizaciones de un fichero. **Cualquier actualización de datos en un recurso compartido, necesitará establecer una región crítica que implicará como mínimo estas instrucciones:**

- **Leer el dato que se quiere actualizar.** Pasar el dato a la zona de memoria local al proceso.
- **Realizar el cálculo de actualización.** Modificar el dato en memoria.

- **Escribir el dato actualizado.** Llevar el dato modificado de memoria al recurso compartido.

Ejemplo: Acceso a ficheros con sincronización.

Procedimiento habitual:

1. Hacer una referencia con `File` al fichero que queremos bloquear.
2. Acceder al fichero utilizando **RandomAccessFile** en modo lectura y escritura. Sobre este objeto, obtener su canal de acceso con `getChannel()`

```
FileChannel channel = new RandomAccessFile(file, "rw").getChannel();
```

Lo que manejamos en este momento es un objeto del tipo **FileChannel**.

El caso del **FileChannel** se representa una conexión abierta sobre un recurso sobre la cual se pueden realizar múltiples operaciones. Una de las operaciones que podemos realizar sobre el canal es bloquearlo o liberarlo.

3. Bloquear el fichero con el método `lock()` sobre un objeto de la clase **FileLock**

```
FileLock lock = channel.lock();
```

Lo que obtenemos en este caso es un objeto **FileLock**. Este objeto será el que nos sirva para chequear el estado del canal/fichero antes de acceder a él.

Si queremos ver si el fichero está bloqueado utilizamos el método `tryLock()` que devolverá una excepción *OverlappingFileLockException* en el caso de que haya un bloqueo. Por ejemplo:

```
1. try{
2. lock = channel.tryLock();
3. }catch(OverlappingFileLockException e){}
```

Para liberar el bloqueo, se usa el método `release()`

```
lock.release();
```

4. Lo último que deberemos de hacer a la hora de bloquear un fichero con Java es cerrar el canal sobre el fichero con el método `.close()`

```
channel.close();
```

Recordatorio: Clase RandomAccessFile

La clase Java **RandomAccessFile** se utiliza para acceder a un fichero de forma aleatoria, para poder acceder a distintas partes de un fichero como accedemos a distintos registros en una base de datos. Los constructores de la clase son:

```
RandomAccessFile(String path, String modo);
```

```
RandomAccessFile(File objetoFile, String modo);
```

Lanzan una excepción `FileNotFoundException`.

El argumento modo indica el modo de acceso en el que se abre el fichero. Los valores permitidos para este parámetro son:

- "r": Abre el fichero en modo solo lectura. El fichero debe existir. Una operación de escritura en este fichero lanzará una excepción IOException.
- "rw": Abre el fichero en modo lectura y escritura. Si el fichero no existe se crea.
- "rwd": Abierto para la lectura y la escritura, al igual que con " rw " y también requieren que cada actualización para el contenido del archivo se escribirá de forma sincronizada con el dispositivo de almacenamiento subyacente

El modo " RWD " se puede utilizar para reducir el número de operaciones de E / S realizadas. El uso de " RWD " sólo requiere cambios a el contenido del archivo que se escriben en el almacenamiento ; el uso de " RW " requiere actualizaciones de los contenidos tanto del archivo y sus metadatos a ser escritas , que generalmente requiere al menos uno más operación de E / S de bajo nivel

7. Documentación.

Para hacer la documentación tradicionalmente hemos usado **JavaDOC**. Sin embargo, las versiones más modernas de Java incluyen las anotaciones.

Una anotación es un texto que pueden utilizar otras herramientas (no solo el Javadoc) para comprender mejor qué hace ese código o como documentarlo.

Cualquiera puede crear sus propias anotaciones simplemente definiéndolas como un interfaz Java. Sin embargo, tendremos que programar nuestras propias clases para extraer la información que proporcionan dichas anotaciones.

8. Depuración.

¿Como se depura un programa multiproceso/multihilo? Por desgracia puede ser muy difícil:

1. No todos los depuradores son capaces.
2. A veces cuando un depurador interviene en un proceso puede ocurrir que el resto de procesos consigan ejecutarse en el orden correcto y dar lugar a que el programa parezca que funciona bien.
3. Un error muy típico es la `NullPointerException`, que en muchos casos se deben a la utilización de referencias Java no inicializadas o incluso a la devolución de valores NULL que luego no se comprueban en alguna parte del código.
4. Se puede usar el método `redirectError` pasándole un objeto de tipo `File` para que los mensajes de error vayan a un fichero.
5. Se debe recordar que la «visión» que tiene Netbeans del sistema puede ser **muy diferente** de la visión que tiene el proceso lanzado. Un problema muy común es que el proceso lanzado no encuentre clases, lo que obligará a indicar el `CLASSPATH`.

6. Un buen método para determinar errores consiste en utilizar el entorno de consola para lanzar comandos para ver «cómo es el sistema» que ve un proceso fuera de Netbeans (o de cualquier otro entorno).