# MÓDULO PROGRAMACIÓN DE SERVICIOS Y PROCESOS

## TEMA 1

## PROGRAMACIÓN MULTIPROCESO

CICLO DE GRADO SUPERIOR INFÓRMATICA

DESARROLLO DE APLICACIONES MULTIPLATAFORMA

**Autor: Luis Miguel Lestón López a partir de los materiales generados por el Ministerio de Educación para este módulo. Obra derivada.**

# ÍNDICE

# 1. Introducción: Aplicaciones, Ejecutables y Procesos.

A simple vista, parece que con los términos aplicación, ejecutable y proceso, nos estamos refiriendo a lo mismo. Pero, no olvidemos que en los módulos de primero hemos aprendido a diferenciarlos.

Una **aplicación** es un tipo de programa informático, diseñado como herramienta para resolver de manera automática un problema específico del usuario.

Debemos darnos cuenta de que sobre el hardware del equipo, todo lo que se ejecuta son programas informáticos, que, ya sabemos, que se llama software. Con la definición de aplicación anterior, buscamos diferenciar las aplicaciones, de otro tipo de programas informáticos, como pueden ser: los sistemas operativos, las utilidades para el mantenimiento del sistema, o las herramientas para el desarrollo de software. Por lo tanto, son aplicaciones, aquellos programas que nos permiten editar una imagen, enviar un correo electrónico, navegar en Internet, editar un documento de texto, chatear, etc.

Recordemos, que un programa es el conjunto de instrucciones que ejecutadas en un ordenador realizarán una tarea o ayudarán al usuario a realizarla. Nosotros, como programadores y programadoras, creamos un programa, escribiendo su código fuente; con ayuda de un compilador, obtenemos su código binario o interpretado. Este código binario o interpretado, lo guardamos en un fichero. Este fichero, es un fichero ejecutable, llamado comúnmente: ejecutable o binario.

Un **ejecutable** es un fichero que contiene el código binario o interpretado que será ejecutado en un ordenador.

De forma sencilla, un **proceso**, es un programa en ejecución. Pero, es más que eso, un proceso en el sistema operativo (SO), es una unidad de trabajo completa; y, el SO gestiona los distintos procesos que se encuentren en ejecución en el equipo. En siguientes apartados de esta unidad trataremos más en profundidad todo lo relacionado con los procesos y el SO. Lo más importante, es que diferenciemos que un ejecutable es un fichero y un proceso es una entidad activa, el contenido del ejecutable, ejecutándose.

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

Un proceso existe mientras que se esté ejecutando una aplicación. Es más, la ejecución de una aplicación, puede implicar que se arranquen varios procesos en nuestro equipo; y puede estar formada por varios ejecutables y librerías.

Una **aplicación** es un tipo de programa informático, diseñado como herramienta para resolver de manera automática un problema específico del usuario. Al instalarla en el equipo, podremos ver que puede estar formada por varios ejecutables y librerías. Siempre que lancemos la ejecución de una aplicación, se creará, al menos, un proceso nuevo en nuestro sistema.

## 1.1. Ejecutables. Tipos

En sistemas operativos Windows, podemos reconocer un fichero ejecutable, porque su extensión, suele ser .exe. En otros sistemas operativos, por ejemplo, los basados en GNU/Linux, los ficheros ejecutables se identifican como ficheros que tienen activado su permiso de ejecución (y no tienen que tener una extensión determinada).

Según el tipo de código que contenga un ejecutable, los podemos clasificar en:

***Binarios***. Formados por un conjunto de instrucciones que directamente son ejecutadas por el procesador del ordenador. Este código se obtiene al compilar el código fuente de un programa y se guarda en un fichero ejecutable. Este código sólo se ejecutará correctamente en equipos cuya plataforma sea compatible con aquella para la que ha sido compilado (no es multiplataforma). Ejemplos son, ficheros que obtenemos al compilar un ejecutable de C o C++.

***Interpretados.*** Código que suele tratarse como un ejecutable, pero no es código binario, sino otro tipo de código, que en Java, por ejemplo se llama bytecode. Está formado por códigos de operación que tomará el intérprete (en el caso de Java, el intérprete es la máquina virtual Java o JRE). Ese intérprete será el encargado de traducirlos al lenguaje máquina que ejecutará el procesador. El código interpretado es más susceptible de ser multiplataforma o independiente de la máquina física en la que se haya compilado.

Un tipo especial de ejecutables interpretados, son los llamados scripts. Estos ficheros, contienen las instrucciones que serán ejecutadas una detrás de otra por el intérprete. Se diferencian de otros lenguajes interpretados porque no son compilados. Por lo que los podremos abrir y ver el código que contienen con un editor de texto plano (cosa que no pasa con los binarios e interpretados compilados). Los intérpretes de este tipo de lenguajes se suelen llamar motores. Ejemplos de lenguajes de script son: JavaScript, php, JSP, ASP, python, ficheros .BAT en MS-DOS, Powershell en Windows, bash scripts en GNU/Linux, …

***Librerías.*** Conjunto de funciones que permiten dar modularidad y reusabilidad a nuestros programas. Las hemos incluido en esta clasificación, porque su contenido es código ejecutable, aunque ese código sea ejecutado por todos los programas que invoquen las funciones que contienen. El conjunto de funciones que incorpora una librería suele ser altamente reutilizable y útil para los programadores; evitando que tengan que reescribir una y otra vez el código que realiza la misma tarea. Ejemplo de librerías son: las librerías estándar de C, los paquetes compilados DLL en Windows; las API (Interfaz de Programación de Aplicaciones), como la J2EE de Java (Plataforma Java Enterprise Edition versión 2); las librerías que incorpora el framework de .NET; etc.

# 2. Gestión de procesos

Como sabemos, en nuestro equipo, se están ejecutando al mismo tiempo, muchos procesos. Por ejemplo, podemos estar escuchando música con nuestro reproductor multimedia favorito; al mismo tiempo, estamos programando con Eclipse; tenemos el navegador web abierto, para ver los contenidos de esta unidad; incluso, tenemos abierto el Skype para chatear con nuestros amigos y amigas.

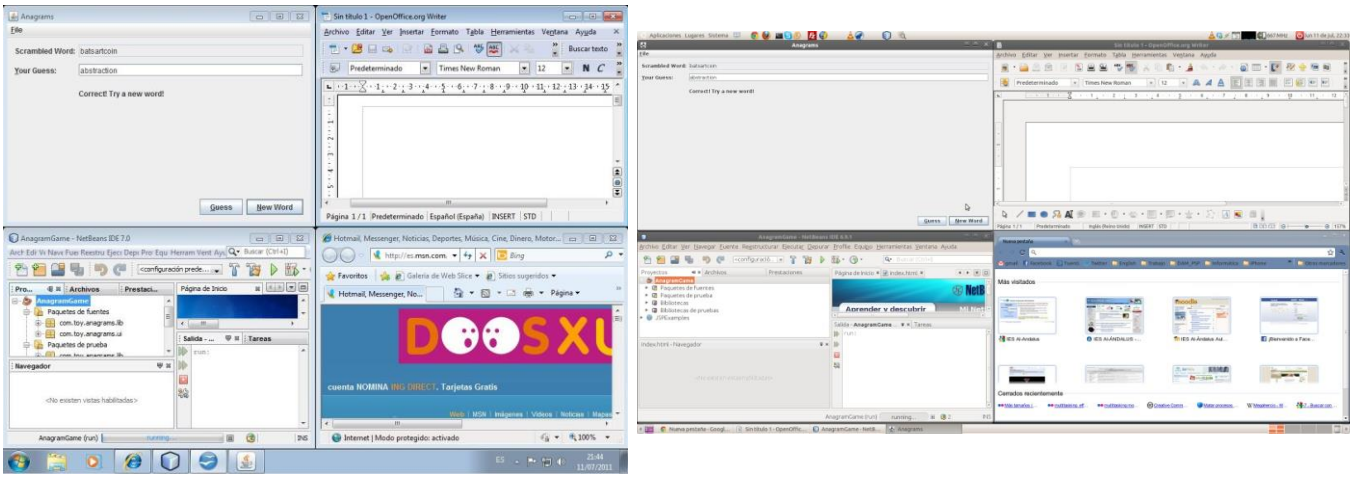
Independientemente de que el microprocesador de nuestro equipo sea más o menos moderno (con uno o varios núcleos de procesamiento), lo que nos interesa es que actualmente, nuestros SO son **multitarea**; como son, por ejemplo, Windows y GNU/Linux. Ser multitarea es, precisamente, permitir que varios procesos puedan ejecutarse al mismo tiempo, haciendo que todos ellos compartan el núcleo o núcleos del procesador. Pero, ¿cómo? Imaginemos que nuestro equipo, es como nosotros mismos cuando tenemos más de una tarea que realizar. Podemos, ir realizando cada tarea una detrás de otra, o, por el contrario, ir realizando un poco de cada tarea. Al final, tendremos realizadas todas las tareas, pero para otra persona que nos esté mirando desde fuera, le parecerá que, de la primera forma, vamos muy lentos (y más, si está esperando el resultado de una de las tareas que tenemos que realizar); sin embargo, de la segunda forma, le parecerá que estamos muy ocupados, pero que poco a poco estamos haciendo lo que nos ha pedido. Pues bien, el micro, es nuestro cuerpo, y el SO es el encargado de decidir, por medio de la gestión de procesos, si lo hacemos todo de golpe, o una tarea detrás de otra.

En este punto, es interesante que hagamos una pequeña clasificación de los **tipos de procesos** que se ejecutan en el sistema:

- **Por lotes.** Están formados por una serie de tareas, de las que el usuario sólo está interesado en el resultado final. El usuario, sólo introduce las tareas y los datos iniciales, deja que se realice todo el proceso y luego recoge los resultados. Por ejemplo: enviar a imprimir varios documentos, escanear nuestro equipo en busca de virus,...
- **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. Por ejemplo: un procesador de textos; una aplicación formada por formularios que permiten introducir datos en una base de datos; ...
- **Tiempo real.** Tareas en las que es crítico el tiempo de respuesta del sistema. Por ejemplo: el ordenador de a borde de un automóvil, reaccionará ante los eventos del vehículo en un tiempo máximo que consideramos correcto y aceptable. Otro ejemplo, son los equipos que controlan los brazos mecánicos en los procesos industriales de fabricación.

## 2.1. Gestión de procesos. Introducción

En nuestros equipos ejecutamos distintas aplicaciones interactivas y por lotes. Como sabemos, un microprocesador es capaz de ejecutar miles de millones de instrucciones básica s en un segundo (por ejemplo, un i7 puede llegar hasta los 3,4 GHz). Un micro, a esa velocidad, es capaz de realizar muchas tareas, y nosotros (muy lentos para él), apreciaremos que solo está ejecutando la aplicación que nosotros estamos utilizando. Al fin y al cabo, al micro, lo único que le importa es ejecutar instrucciones y dar sus resultados, no tiene conocimiento de si pertenecen a uno u otro proceso, para él son instrucciones. Es, el SO el encargado de decidir qué proceso puede entrar a ejecutarse o debe esperar. Lo veremos más adelante, pero se trata de una fila en la que cada proceso coge un número y va tomando su turno de servicio durante un periodo de tiempo en la CPU; pasado ese tiempo, vuelve a ponerse al final de la fila, esperando a que llegue de nuevo su turno.



Vamos a ver cómo el SO es el encargado de gestionar los procesos, qué es realmente un programa en ejecución, qué información asocia el SO a cada proceso. También veremos qué herramientas tenemos a nuestra disposición para poder obtener información sobre los procesos que hay en ejecución en el sistema y qué uso están haciendo de los recursos del equipo.

Los nuevos micros, con varios núcleos, pueden, casi totalmente, dedicar una CPU a la ejecución de uno de los procesos activos en el sistema. Pero no nos olvidemos de que además de estar activos los procesos de usuario, también se estará ejecutando el SO, por lo que seguirá siendo necesario repartir los distintos núcleos entre los procesos que estén en ejecución.

## 2.2. Estados de un Proceso

Si el sistema tiene que repartir el uso del microprocesador entre los distintos procesos, ¿qué le sucede a un proceso cuando no se está ejecutando? Y, si un proceso está esperando datos, ¿por qué el equipo hace otras cosas mientras que un proceso queda a la espera de datos?

Veamos con detenimiento, cómo es que el SO controla la ejecución de los procesos. Ya comentamos en el apartado anterior, que el SO es el encargado de la gestión de procesos. En el siguiente gráfico, podemos ver un esquema muy simple de cómo podemos planificar la ejecución de varios procesos en una CPU.

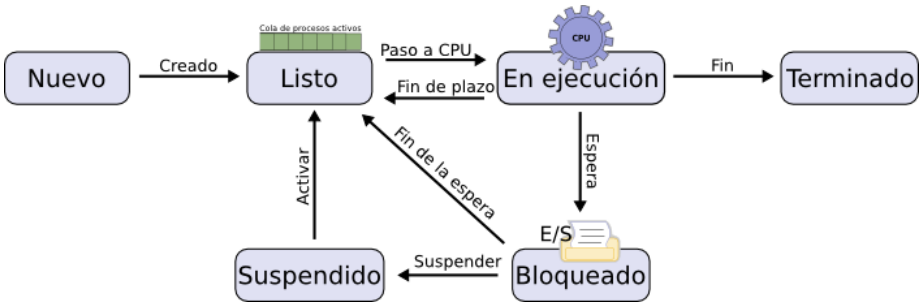

En este esquema, podemos ver:

- Los procesos nuevos, entran en la cola de procesos activos en el sistema.
- Los procesos van avanzando posiciones en la cola de procesos activos, hasta que les toca el turno para que el SO les conceda el uso de la CPU.
- El SO concede el uso de la CPU, a cada proceso durante un tiempo determinado y equitativo, que llamaremos quantum. Un proceso que consume su quantum, es pausado y enviado al final de la cola.
- Si un proceso finaliza, sale del sistema de gestión de procesos.

Esta planificación que hemos descrito, resulta equitativa para todos los procesos (todos van a ir teniendo su quamtum de ejecución). Pero se nos olvidan algunas situaciones y características de nuestros los procesos:

- Cuando un proceso, necesita datos de un archivo o una entrada de datos que deba suministrar el usuario; o, tiene que imprimir o grabar datos; cosa que llamamos 'el proceso está en una **operación de entrada/salida**' (E/S para abreviar). El proceso, queda **bloqueado hasta que haya finalizado** esa E/S. El proceso es bloqueado, porque, los dispositivos son mucho más lentos que la CPU, por lo que, mientras que uno de ellos está esperando una E/S, **otros procesos pueden** pasar a la CPU y **ejecutar sus instrucciones**. Cuando termina la E/S que tenga un proceso bloqueado, el SO, volverá a pasar al proceso a la cola de procesos activos, para que recoja los datos y continúe con su tarea (dentro de sus correspondientes turnos).

- Sólo mencionar (o recordar), que cuando la memoria RAM del equipo está llena, algunos procesos deben pasar a disco (o almacenamiento secundario) para dejar espacio en RAM que permita la ejecución de otros procesos.



Todo proceso en ejecución, tiene que estar cargado en la RAM física del equipo o memoria principal, así como todos los datos que necesite.

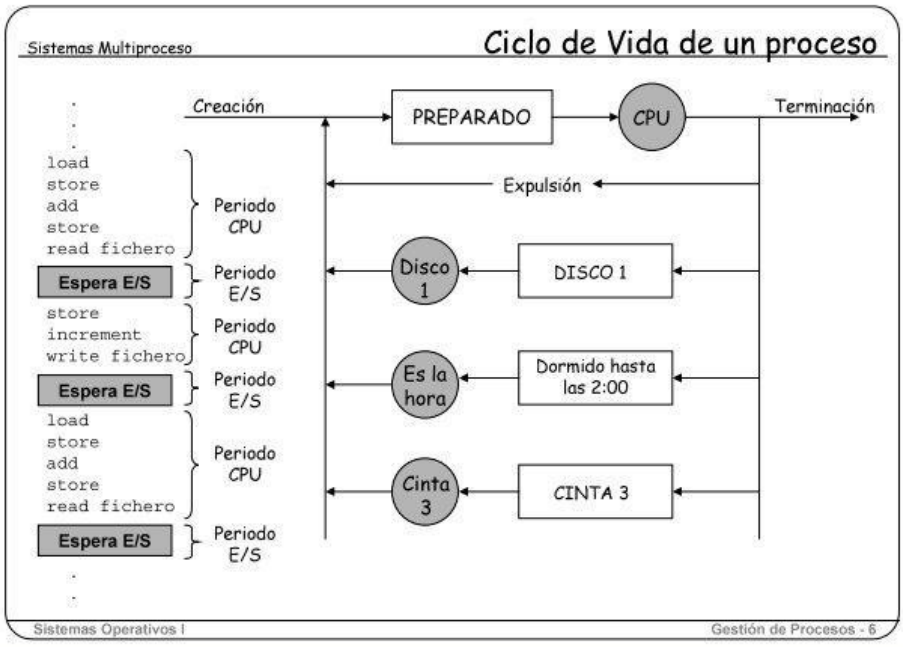
Hay procesos en el equipo cuya ejecución es crítica para el sistema, por lo que, no siempre pueden estar esperando a que les llegue su turno de ejecución, haciendo cola. Por ejemplo, el propio SO es un programa, y por lo tanto un proceso o un conjunto de procesos en ejecución. Se le da prioridad, evidentemente, a los procesos del SO, frente a los procesos de usuario.

Los estados por los que transita un proceso son:

- **En creación**: El proceso está siendo creado
- **En Ejecución:** Se están ejecutando las instrucciones.
- **En espera**: El proceso está esperando a que se produzca un suceso
- **Preparado**: El proceso está a la espera de que se le asigne a un procesador
- **Terminado**: Ha finalizado su ejecución

En el estado en ejecución solo puede haber un proceso, sin embargo, varios pueden estar listos o bloqueados.

Al conjunto de estados por los que transita un proceso se denomina ciclo de vida de un proceso. Este ciclo de vida de un proceso lo podemos ver en la Figura siguiente.

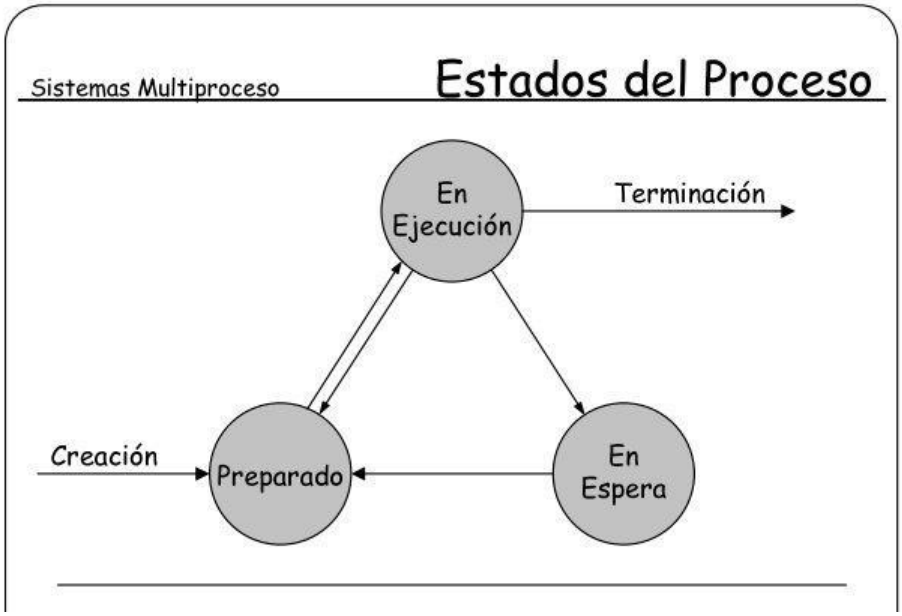



En esta figura, los círculos son recursos o condiciones deseadas por un proceso, mientras que los rectángulos representan las colas de procesos que esperan por tales recursos. Así, vemos que un proceso recién creado inmediatamente quiere ejecutarse, para lo cual necesita la CPU, por lo que pasa a la cola de los procesos que esperan por el procesador (cola de procesos preparados). Una vez que le llega el turno, ejecuta las instrucciones, hasta que por uno de los motivos mencionados

(operación de E/S, fin de la porción de tiempo, etc.) pierde la posesión de la CPU y pasa a la cola del recurso solicitado (Disco 1, Unidad de Cinta 3, esperar a las 2:00 requiere el procesador, el recurso que necesita el proceso expulsado (o expropiado) es la propia CPU, por lo que pasa a la cola correspondiente (la de los procesos preparados).

Este ciclo se repite una y otra vez hasta que el proceso ejecuta su última instrucción, o sea, que finaliza el trabajo, con lo que voluntariamente cede el control de la CPU y da por concluida su existencia como proceso.

Así, llegamos ya al diagrama básico de estados de un proceso, que lo vemos representado en la parte superior de la Figura inferior. En él tenemos que un proceso está ejecutando instrucciones (En Ejecución), esperando a que le concedan la CPU




Los procesos no pueden pasar por ellos mismos de listos a ejecución, es el S.O. el que decide cuando se pasa de listo a ejecutado.

Vamos a ver como se producen las transiciones entre estos tres estados.

### *Creación→Preparado*

Inicialmente, al crear un proceso, no puede pasar directamente al estado de Espera, pues no se ha ejecutado ninguna instrucción que así se lo indique; por lo tanto, expresa su necesidad de ejecución entrando en la cola de procesos Preparados.

### *Preparado→En Ejecución*

Una vez que el proceso está preparado, solamente tiene que esperar a que le llegue el turno. Los turnos del procesador se establecen en función de la política de planificación de la CPU. Cuando le llegue el turno tomara posesión del procesador y empezara a ejecutar instrucciones.

Como podemos ver en el grafico, desde el estado de Ejecución se puede pasar a cualquiera

de los otros dos estados (Espera y Preparado). de la mañana). Si la posesión del procesador se ha perdido por finalizar la porción de tiempo asignado o bien porque un proceso de mayor prioridad

### *En Ejecución→Preparado*

Pasará a Preparado si abandona la CPU involuntariamente, es decir si se le expulsa (o expropia), bien porque haya terminado su porción de tiempo, o porque haya pasado a Preparado otro proceso de mayor prioridad.

### *En Ejecución→Espera*

En cambio, pasa al estado de Espera si cede voluntariamente el procesador, por tener que esperar a que se produzca un evento externo a la CPU.

### *Espera→Preparado*

Desde el estado de Espera no se puede volver directamente a Ejecución, sino que cuando se haya cumplido el evento externo, para recuperar la posesión del procesador hay que pasar de nuevo por la cola de los procesos preparados. Así tenemos que para conseguir la posesión de la CPU siempre hay que pasar por su cola correspondiente (como un recurso más que es).

### *En Ejecución→Terminación*

Como ya hemos dicho antes, el proceso termina cuando ejecuta su última instrucción, luego siempre pasará al estado de Terminado desde Ejecución

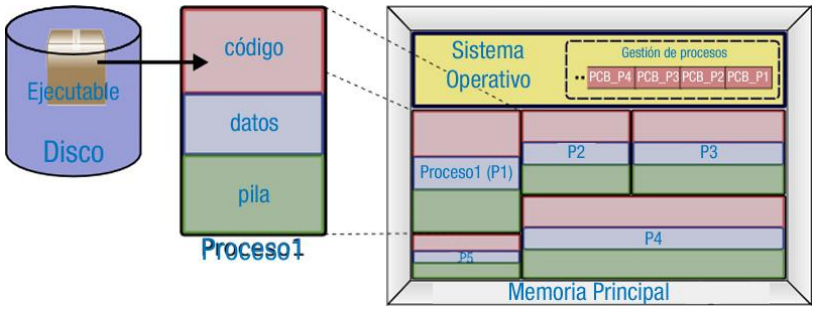
Como hemos visto, a medida que progresa su ejecución, un proceso va cambiando de estado. A excepción de los estados obvios de *Creación* y *Terminación*, un proceso tiene dos estados básicos: *Activo* y *En Espera .*

- **Activo:** El proceso no depende de ninguna condición externa al procesador para poder continuar su ejecución. O bien se está ejecutando, o se encuentra esperando en la cola de procesos Preparados a que le concedan la CPU.
- **En Espera**: El proceso no puede continuar la ejecución. Está bloqueado, a la espera de que se cumpla un evento externo a la CPU, tal como una respuesta de un dispositivo de E/S, que llegue una hora determinada, etc.

Vistos los estados que puede tener un proceso y las transiciones entre ellos, nos interesa saber como el sistema operativo lleva control de todos los procesos creados, cómo se realizan estas colas de procesos y cómo se sabe qué proceso está en ejecución.

Hay una estructura de datos que contiene toda la información de control que requiere el sistema operativo para ocuparse de la gestión o administración del proceso. Nos estamos refiriendo al Descriptor de Proceso o Bloque de Control de Proceso (BCP). Pues bien, entonces lo que el sistema operativo necesita es mantener la colección de todos los BCP's de los procesos existentes en cada momento. Esta colección de BCP's se puede organizar de distintas maneras: mediante una cola estática, con una cola dinámica, o simplemente utilizando una tabla, es decir, un vector o array de BCP's.

## 2.3. Planificación de procesos por el Sistema Operativo




Entonces, ¿un proceso sabe cuando tiene o no la CPU? ¿Cómo se decide qué proceso debe ejecutarse en cada momento?

Hemos visto que un **proceso**, desde su creación hasta su fin (durante su vida), pasa por muchos estados. Esa **transición de estados, es transparente** para él, todo **lo realiza el SO. Desde el punto de vista de un proceso, él siempre se está ejecutando en la CPU sin esperas.** Dentro de la gestión de procesos vamos a destacar dos componentes del SO que llevan a cabo toda la tarea: el cargador y el planificador.

El **cargador es el encargado de crear los procesos.** Cuando se inicia un proceso (para cada proceso), el cargador, realiza las siguientes tareas:

- **Carga el proceso en memoria principal.** Reserva un espacio en la RAM para el proceso. En ese espacio, copia las instrucciones del fichero ejecutable de la aplicación, las constantes y, deja un espacio para los datos (variables) y la pila (llamadas a funciones). **Un proceso, durante su ejecución, no podrá hacer referencia a direcciones que se encuentren fuera de su espacio de memoria;** si lo intentara, el SO lo detectará y generará una excepción (produciendo, por ejemplo, los típicos pantallazos azules de Windows).
- **Crea una estructura de información** llamada **PCB** (Bloque de Control de Proceso). La información del PCB, es única para cada proceso y permite controlarlo. Esta información, también la utilizará el planificador. Entre otros datos, el PCB estará formado por:

  - **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, quamtum, 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, …

## 2.3.1. Planificación de procesos por el Sistema Operativo (II).

Habiendo visto el ciclo de vida de los procesos, sabemos que un proceso que se está ejecutando, eventualmente, por diversos motivos que se han comentado, abandona la posesión del procesador voluntaria o involuntariamente, por lo que hay que cederle la CPU a un proceso que sea lógicamente ejecutable, es decir, que esté Preparado. Si hay más de un proceso Preparado, el sistema operativo debe decidir cuál de ellos se ejecutará primero. La parte del sistema operativo que le corresponde tomar tal decisión es el Planificador, que seleccionará un proceso basado en una política o algoritmo de planificación.

El estado de Preparado se implementa como una cola de procesos dispuestos a ejecutar instrucciones con ayuda de la CPU. Ahora bien, debemos a aclarar que esta cola no tiene por qué ser una cola FIFO (Primero en Entrar, Primero en Salir), también podría ser una cola ordenada por prioridades, un árbol o, simplemente, una lista desordenada. El Planificador debe seleccionar uno de los procesos de la lista, pero no basándose necesariamente en la propia estructura de la lista. La estructura de la lista únicamente debe ayudar al Planificador a aplicar el algoritmo de planificación.

La planificación. Implica tres tipos de planificación.

- **Planificadores a largo plazo (Planificador de trabajos).-** Decide si se añade al conjunto de programas a ser ejecutados.
- **Planificador a mediano plazo.-** Decide si se añade al número de procesos que están total o parcialmente en memoria principal.
- **Planificadores a corto plazo (Planificador del CPU).-** Decide cuál de los procesos disponibles ejecutará el proceador.

### Criterios de planificación a corto plazo

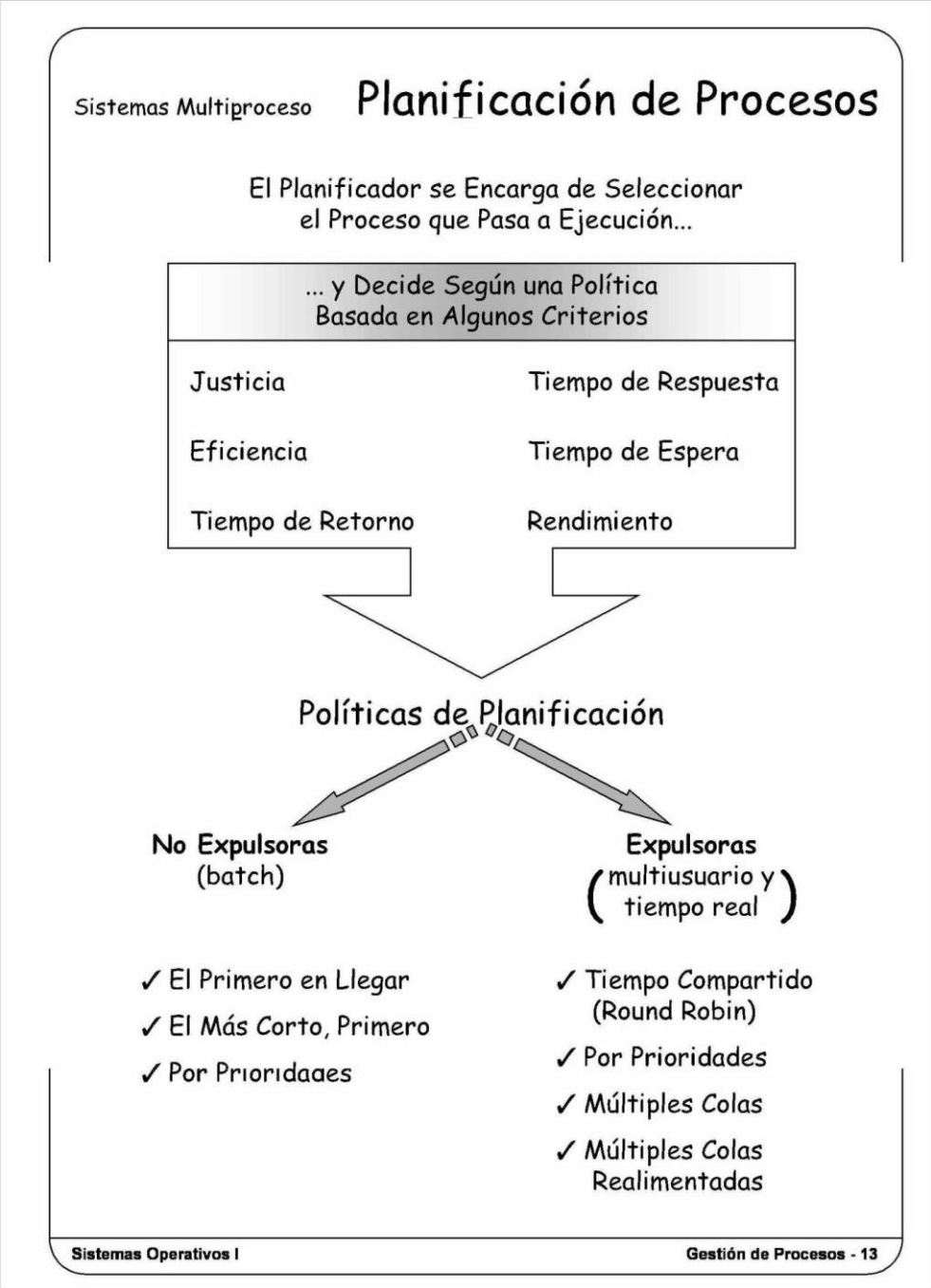
Antes de ver algoritmos concretos de planificación debemos recordar, que la elección que el Planificador va a realizar debe hacerse persiguiendo "el bien del sistema", pues es uno de los cometidos del sistema operativo. No obstante, distintas políticas de planificación tienen diferentes propiedades, y favorecen más a un tipo de procesos que a otros. Antes de elegir el algoritmo a utilizar en una situación concreta, debemos considerar las propiedades de varios algoritmos.

Se han sugerido muchos criterios para comparar algoritmos de planificación de CPU. Decidir las características que se van a utilizar en la comparación es fundamental para la elección del algoritmo más apropiado. Veamos los criterios de comparación más comúnmente utilizados:

- Justicia. Cada proceso debe conseguir su porción correspondiente de CPU en un tiempo finito.

- Eficiencia. Se debe intentar mantener la CPU ocupada el mayor tiempo posible. Al decir "ocupada" queremos decir ejecutando cualquier proceso que no sea el Proceso Ocioso. En un sistema real, el porcentaje de utilización de CPU suele estar en el rango 40-90%. Una utilización mayor del 90% significaría que el Planificador siempre encuentra procesos en la cola Preparados, o sea, que dicha cola suele contener un número considerable de procesos. En un sistema interactivo, esto puede suponer que los usuarios quizás se están impacientando por obtener las respuestas.
- Tiempo de Retorno (*turnaround time*). Desde el punto de vista de un proceso, el criterio más importante es cuánto tiempo se va a necesitar para ejecutarse completamente. El Tiempo de Retorno es la suma de los periodos que se pasan esperando a cargarse en memoria, esperando en la cola de Preparados, ejecutándose en la CPU, y esperando por operaciones de E/S. Así pues, se debe minimizar el tiempo de retorno de un proceso. Afecta principalmente a los procesos *batch*.
- Tiempo de Espera. El algoritmo de planificación de CPU no afecta a la cantidad de tiempo que un proceso se pasa realizando operaciones de E/S, solamente afecta al tiempo que un proceso se pasa en la cola Preparados. El Tiempo de Espera es la suma de todos los momentos que un proceso pasa en la cola de los procesos preparados.
- Tiempo de Respuesta. En un sistema interactivo, el Tiempo de Retorno puede no ser un buen criterio. A menudo, un proceso puede producir algunos resultados al principio, y puede continuar calculando nuevos resultados mientras los anteriores se le muestran al usuario. Así, tenemos que otra medida es el tiempo que transcurre desde que se le hace una petición al sistema hasta que empieza a responder, sin tener en cuenta el tiempo que se tarda en mostrar la respuesta completa.
- Rendimiento. Se debe maximizar el número de trabajos procesados por unidad de tiempo.

Es deseable maximizar la utilización de la CPU y minimizar los tiempos de retorno, de espera y de respuesta, todo ello con la mayor justicia para todos los procesos. Sin embargo, es fácil observar que algunos de estos criterios son contradictorios. Por ejemplo, para que en los procesos interactivos el tiempo de respuesta sea bueno, se puede impedir que se ejecuten procesos batch por el día, reservando para éstos las horas nocturnas en las que no suele haber usuarios en los terminales. Esto no les sentará muy bien a los usuarios que han encargado trabajos en batch, pues ven que el tiempo de retorno se incrementa. Como en otros aspectos de la vida, lo que beneficia a unos, perjudica a otros; así que, en cualquier caso, no habrá que olvidarse nunca del criterio que hace referencia a la justicia.




En pro de la justicia, en la mayoría de los casos se suelen optimizar los valores medios, no obstante, bajo algunas circunstancias, es deseable optimizar los mínimos o los máximos valores, en lugar de la media. Por ejemplo, para garantizar que todos los usuarios obtienen un buen servicio, lo que se desea es minimizar el tiempo máximo de respuesta.

También se ha sugerido que, en los sistemas interactivos, es más importante minimizar la varianza en el tiempo de respuesta, que minimizar su valor medio. Es preferible un sistema con un tiempo de respuesta razonable y predecible, que otro que aunque por término medio resulte más rápido, sea altamente variable.

## Políticas de Planificación

En los diagramas de estados de los procesos, vimos que cuando un proceso en Ejecución abandona tal estado pasa a Espera o a Preparado, dependiendo si deja el procesador voluntaria o involuntariamente. Si los procesos de un sistema nunca dejan la CPU de forma involuntaria, se dice que la política de planificación de CPU es no expulsora o no expropiativa (*non preemptive scheduling*). Por el contrario, si pueden perder la posesión del procesador sin solicitarlo, nos encontramos con una planificación expulsora o expropiativa (*preemptive scheduling*).

Las políticas no expulsoras suelen aplicarse en sistemas batch o en pequeños entornos monousuario, como el sistema operativo MS-DOS o el entorno Windows en todas sus versiones para usuarios domésticos, ambos de Microsoft. Algunos algoritmos que se emplean en esta planificación son *primero en llegar - primero en servir*, *el mas corto primero*, y *por prioridades*.

Por otra parte, los algoritmos expropiativos se utilizan en sistemas de control industrial y entornos multiusuario. Los algoritmos más utilizados aquí incluyen *Round-Robin*, *por prioridades*, de *multiples colas* y de *múltiples colas realimentadas* (como es el caso de Unix y Windows Server).

Veamos a continuación algunos comentarios sobre estos algoritmos.

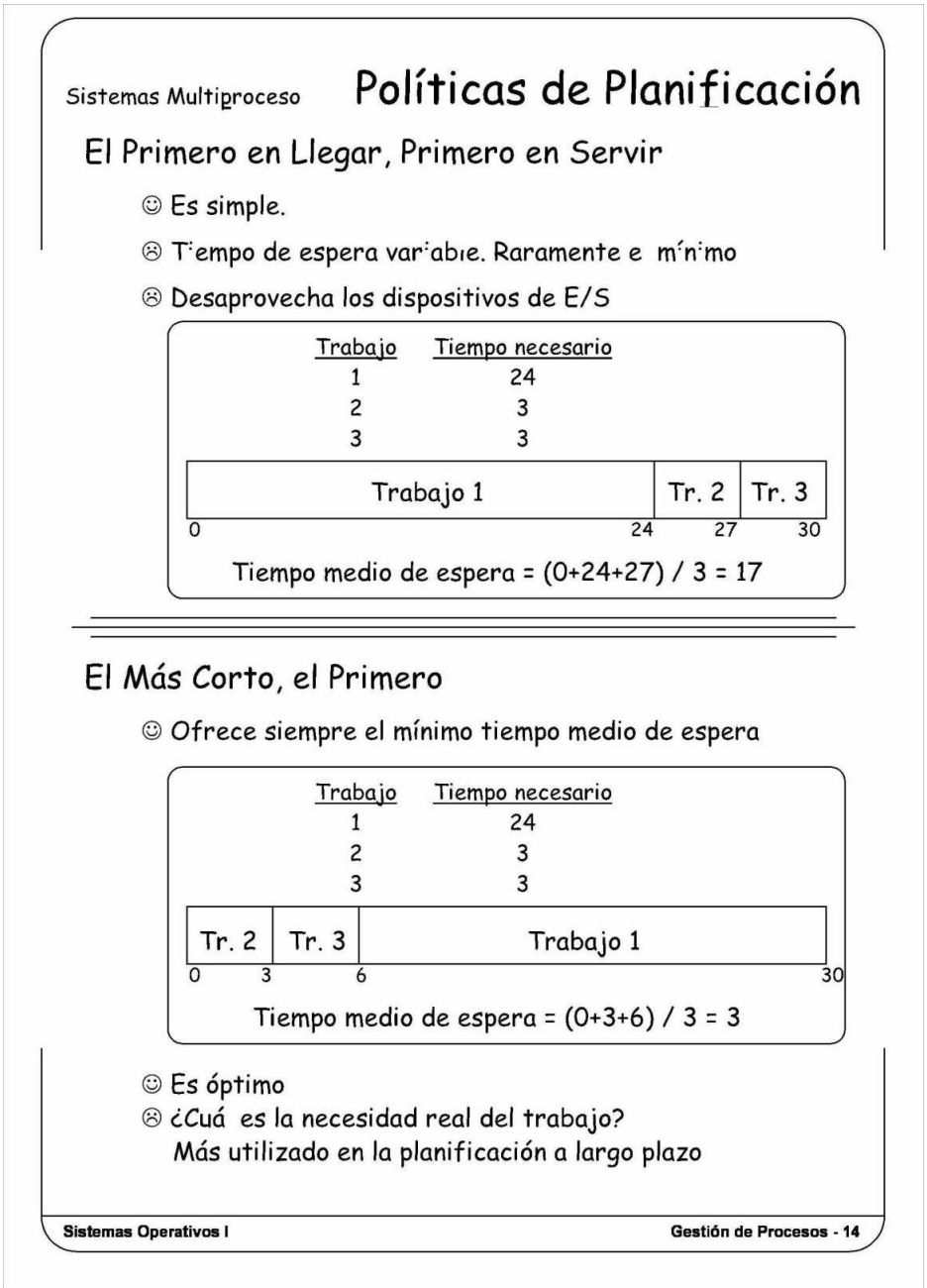
- **Primero en llegar - Primero en servir**

Este algoritmo, cuyo nombre abreviaremos con FCFS (*First Come - First Served*), es, con mucho, el algoritmo más simple. Según este esquema, el primer proceso que reclama la CPU, es el primero en conseguirla. En cuanto a la implementación, no merece la pena dar muchas explicaciones, pues basta con mantener, por ejemplo, una cola FIFO de los descriptores de los procesos que van solicitando la CPU.

El tiempo medio de espera es muy variable y raramente es el mínimo. En el gráfico superior de la Figura 14 se muestra un caso que según ese orden de llegada, resulta un tiempo medio de espera de 17 ms. En cambio, si el orden de llegada de los trabajos o procesos hubiera sido T2, T3, T1, se puede comprobar que habría dado lugar a un tiempo medio de espera de 3 ms.

Otro problema que puede aparecer al aplicar este algoritmo se da en el siguiente escenario. Supongamos un sistema con esta política FCFS en el que hay un proceso (Proceso-CPU) que consume grandes porciones de CPU y realiza pocas operaciones de E/S. Por el contrario, hay muchos otros procesos con un ciclo en el que realizan frecuentes operaciones de E/S y ejecutan muy pocas instrucciones sobre la CPU (Procesos-E/S). Mientras el Proceso-CPU se ejecuta, los Procesos-E/S acabarán sus operaciones con los dispositivos periféricos y pasarán a la cola Preparados, quedando libres los dispositivos de E/S. En algún momento, el Proceso-CPU acabará por realizar una operación de E/S y pasará a Espera. Los Procesos-E/S, que necesitan poca CPU se ejecutan rápidamente y vuelven a las colas de espera de los periféricos de E/S. La CPU queda ociosa. El Proceso-CPU finaliza la operación de E/S y vuelve a tomar control del procesador. Otra vez, los Procesos-E/S terminarán sus operaciones de E/S

y pasarán a Preparados, esperando todos a que el Proceso-CPU realice otra operación de E/S. Cuando se produce esta situación, en la que muchos procesos que necesitan poca CPU esperan por uno que consume mucho tiempo de procesador, se denomina *efecto convoy*. Este efecto provoca una menor utilización de la CPU y de los periféricos de la que se podría conseguir si se concediese la posesión de la CPU a los procesos que requieren poco tiempo de procesador, en lugar de cedérselo a un proceso que va a hacer esperar mucho tiempo a muchos procesos.



- **El más corto primero**

Este algoritmo, también conocido como SJF (*Shortest Job First*), concede la CPU al proceso que durante menos tiempo necesita la CPU de forma ininterrumpida. Debe quedar claro que se selecciona el que menor porción de tiempo de CPU necesita en el momento de hacer la elección, no aquel cuyo tiempo total de CPU es el menor de todos. Si dos procesos necesitan el procesador durante el mismo tiempo, se elige uno mediante FCFS.

En la parte inferior de la Figura 14 tenemos la aplicación de este algoritmo a nuestro ejemplo. Como se puede ver, consigue el menor tiempo de espera posible. Poniendo a un proceso corto por delante de uno largo, se decremento el tiempo de espera del corto más de lo que se incrementa el del largo. Consecuentemente, el tiempo medio decrece.

Este algoritmo, probablemente el Optimo, se comporta bien en la planificación a largo plazo de los procesos batch, en los que se conocen los requisitos de tiempo total de CPU indicados por el usuario; pero en la planificación de la CPU se presenta el problema de que en el momento en que hay que asignar la CPU, no hay forma de conocer a priori la porción de tiempo que necesita cada proceso, por lo que es difícilmente implementable. Se puede intentar una aproximación al SJF puro, haciendo una estimación de la porción necesaria basándose en estadísticas de ejecuciones anteriores.

- ***Round - Robin***

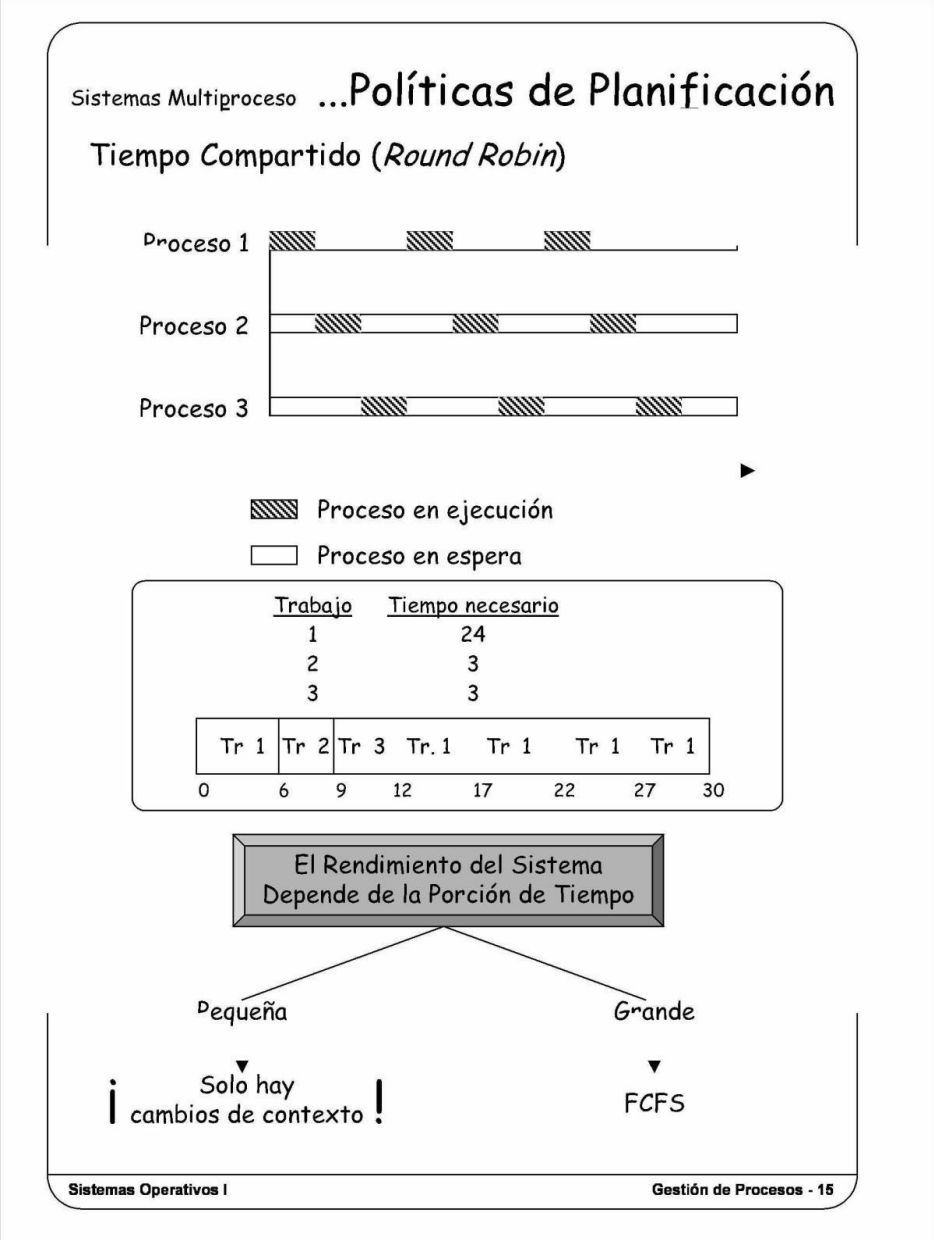
Este algoritmo esta especialmente diseñado para los sistemas multiusuario de tiempo compartido. Es similar al FCFS, pero se le añade la expropiación de la CPU cuando la posesión continuada de esta sobrepasa un tiempo preestablecido (porción de tiempo), que suele ser del orden de 10 a 100 milisegundos.

La implementación de este algoritmo se realiza con una cola FIFO para los procesos Preparados. Los nuevos procesos se añaden al final de la cola. Cuando el planificador tiene que seleccionar un proceso, elige el primero de la cola, establece una temporización correspondiente a la porción de tiempo, y le cede el control. Ahora pueden ocurrir dos cosas:

> 1a) El proceso termina o realiza una operación de E/S antes de que se acabe su porción de tiempo, en cuyo caso cede la CPU voluntariamente, pasa a espera y el planificador selecciona el siguiente proceso de la cola Preparados.
>
> 2a) El proceso se ejecuta hasta dar lugar a que venza la temporización establecida. En este caso, el dispositivo temporizador genera una interrupción que captura el sistema operativo, dando lugar a que se le expropie la CPU al proceso en ejecución, que pasa al final de la cola Preparados, y a que el planificador seleccione al primer proceso de la cola para pasar a ejecución.

Ya habíamos comentado anteriormente la falta de eficiencia que pueden causar los

cambios continuos del proceso en ejecución. En el caso de que la política de planificación del procesador sea por Prioridades de los procesos, se nos hace imposible establecer la frecuencia de los cambios, pues vendrá impuesta por los fenómenos externos que afectan a los procesos correspondientes. En cambio, si la planificación es por Tiempo Compartido, si esta en nuestra mano establecer porciones de tiempo de CPU muy grandes, tal que en una unidad de tiempo se produzcan muy pocos cambios del proceso en ejecución. Sin embargo, esto no resulta tan fácil de decidir, pues si la porción de tiempo se hace muy grande, el aprovechamiento de la CPU por parte del usuario se ve altamente incrementada, pero, por contra, también ocurrirá que los usuarios de los terminales notaran que "se les tarda mucho en atender", pues se ha degenerado en una política FCFS. Por el contrario, si la porción se hace muy pequeña, se les atiende enseguida, pero durante muy poco tiempo, con la consiguiente degradación en el aprovechamiento de la CPU por parte del usuario.



- **Por Prioridades**

Uno de los factores a tener en cuenta en la planificación de los procesos es la justicia, pero justicia no siempre significa un reparto a partes iguales, sino la asignación más conveniente que aconsejen las circunstancias. Así, por ejemplo, tenemos que en un programa de control y supervisión de una central nuclear, compuesto por diversos procesos, no se debe esperar que todos los procesos cuenten inexcusablemente con la misma porción de tiempo de CPU, ni según un turno circular. Evidentemente, no es tan importante el proceso interactivo que se encarga de mostrar a los visitantes un video de la historia de la central y responder a sus preguntas, como los procesos encargados de supervisar los niveles de presión, temperatura, radioactividad, etc., y que en caso de que se sobrepasen los límites de seguridad activan las válvulas y avisos correspondientes.

Los criterios para establecer las prioridades pueden ser variados: por razones físicas, como en el caso de los sistemas de tiempo real, por rangos o categorías de los usuarios (sistemas multiusuarios), por factores políticos, etc. En sistemas multiusuario, es muy común tener prioridades por grupos de trabajo o departamentos, y a los procesos de la misma prioridad aplicarles una política reparto de tiempo compartido.

Este algoritmo puede ser *expulsor* o *no expulsor*. Cuando un proceso llega a la cola Preparados, se compara su prioridad con la del proceso en ejecución. Con un algoritmo expulsor por prioridades, si la prioridad del recién llegado es mayor que la del proceso en ejecución, se le expropia la CPU a este ultimo en favor del recién llegado. Si la política es no expulsora, simplemente se coloca al nuevo proceso a la cabeza de la cola. El algoritmo SJF es un caso particular de este algoritmo con política no expulsora, donde la prioridad es la inversa del tiempo requerido de CPU.

- **Multiples colas**

Existe otro tipo de algoritmos para situaciones en las que los procesos están claramente clasificados en diferentes grupos. Por ejemplo, una división muy común es la que se hace entre procesos interactivos de primer plano (*foreground*) y los procesos batch que suelen ejecutarse en un segundo plano (*background*). Estos dos tipos de procesos tiene distintos requisitos en sus tiempos de respuesta, por lo que podrían tener distintos algoritmos de planificación cada uno. Además, los procesos interactivos podrían tener mayor prioridad que los se ejecutan en un segundo plano.

Para esto, la cola Preparados podría estar dividida en varias colas separadas, una por cada tipo de proceso. Cada proceso siempre se encola en su cola correspondiente. Así nos encontramos con que para elegir la cola de la que se va a sacar un proceso se utiliza una política, mientras que para seleccionar qué proceso se va a sacar de la cola elegida, se utiliza un algoritmo distinto. También puede ocurrir que en dos colas distintas se utilice la misma política de planificación, por ejemplo, tiempo compartido, pero que la porción de tiempo sea distinta en una cola que en otra.

## 2.4. Cambio de contexto en la CPU

Un **proceso es una unidad de trabajo completa.** El sistema operativo es el encargado de gestionar los procesos en ejecución de forma eficiente, intentando evitar que haya conflictos en el uso que hacen de los distintos recursos del sistema. Para realizar esta tarea de forma correcta, se asocia a cada proceso un conjunto de información (PCB) y de unos mecanismos de protección (un espacio de direcciones de memoria del que no se puede salir y una prioridad de ejecución).

Imaginemos que, en nuestro equipo, en un momento determinado, podemos estar escuchando música, editando un documento, al mismo tiempo, chateando con otras personas y navegando en Internet. En este caso, tendremos ejecutándose en el sistema cuatro aplicaciones distintas, que pueden ser: el reproductor multimedia VLC, el editor de textos writer de OpenOffice, el Messenger y el navegador Firefox. Todos ellos, ejecutados sin fallos y cada uno haciendo uso de sus datos.

El sistema operativo (el planificador), al realizar el cambio una aplicación a otra, tiene que guardar el estado en el que se encuentra el microprocesador y cargar el estado en el que estaba el microprocesador cuando cortó la ejecución de otro proceso, para continuar con ese. Pero, ¿qué es el **estado de la CPU?**

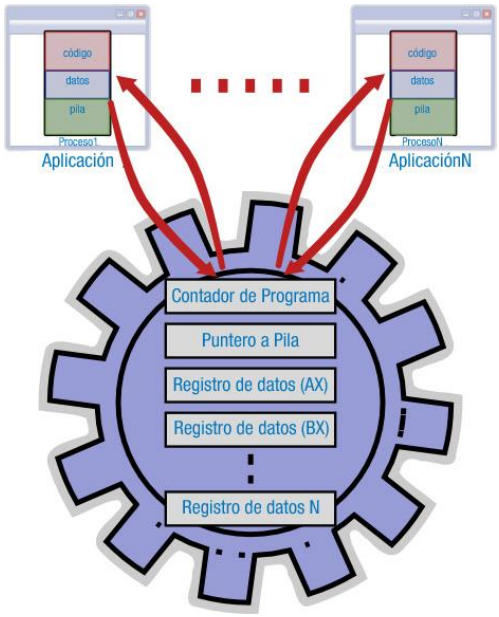
Una CPU, además de circuitos encargados de realizar las operaciones con los datos (llamados circuitos operacionales), tiene unos pequeños espacios de memoria (llamados registros), en los que se **almacenan temporalmente la información que, en cada instante, necesita la instrucción que esté procesando la CPU. El conjunto de registros de la CPU es su estado.**

Entre los registros, destacamos el **Registro Contador de Programa y el puntero a la pila.**

- El **Contador de Programa**, en cada instante almacena la **dirección de la siguiente instrucción a ejecutar**. Recordemos, que cada instrucción a ejecutar, junto con los datos que necesite, es llevada desde la memoria principal a un registro de la CPU para que sea procesada; y, el resultado de la ejecución, dependiendo del caso, se vuelve a llevar a memoria (a la dirección que ocupe la correspondiente variable). Pues el Contador de Programa, apunta a la dirección de la siguiente instrucción que habrá que traer de la memoria, cuando se termine de procesar la instrucción en curso. Este Contador de Programa **nos permitirá continuar en cada proceso por la instrucción en dónde lo hubiéramos** dejado todo.

- El **Puntero a Pila**, en cada instante apunta a la parte superior de la pila del proceso en ejecución. En la pila de cada proceso es donde será almacenado el contexto de la CPU. Y de donde se recuperará cuando ese proceso vuelva a ejecutarse.

La CPU realiza un **cambio de contexto** cada vez que cambia la ejecución de un proceso a otro distinto. En un cambio de contexto, hay que **guardar el estado actual de la CPU y restaurar el estado de CPU del proceso que va a pasar a ejecutar.**




## 2.5. Servicios. Hilos

En este apartado, haremos una breve introducción a los conceptos servicio e hilo, ya que los trataremos en profundidad en el resto de unidades de este módulo.

El ejemplo más claro de **hilo** o **thread**, es un juego. El juego, es la aplicación y, mientras que nosotros controlamos uno de los personajes, los 'malos' también se mueven, interactúan por el escenario y quitan vida. Cada uno de los personajes del juego es controlado por un hilo. Todos los hilos forman parte de la misma aplicación, **cada uno** actúa siguiendo un patrón de comportamiento. El comportamiento es el **algoritmo** que cada uno de ellos seguirá. Sin embargo, todos esos hilos **comparten la información de la aplicación**: el número de vidas restantes, la puntuación obtenida hasta eso momento, la posición en la que se encuentra el personaje del usuario y el resto de personajes, si ha llegado el final del juego, etc. Como sabemos, esas informaciones son variables. Pues bien, un proceso, no puede acceder directamente a la información de otro proceso. Pero, los **hilos** de un mismo proceso están dentro de él, por lo que **comparten la información de las variables de ese proceso.**

**Realizar cambios de contexto entre hilos de un mismo proceso, es más rápido y menos costoso que el cambio de contexto entre procesos,** ya que sólo hay que cambiar el valor del registro contador de programa de la CPU y no todos los valores de los registros de la CPU.

**Un proceso, estará formado por, al menos, un hilo de ejecución.**

**Un proceso es una unidad pesada de ejecución. Si el proceso tiene varios hilos, cada hilo, es una unidad de ejecución ligera.**

Un **servicio** es un proceso que, normalmente, es cargado durante el arranque del sistema operativo. Recibe el nombre de servicio, ya que es un **proceso que queda a la espera de que otro le pida que realice una tarea.** Por ejemplo, tenemos el servicio de impresión con su típica cola de trabajos a imprimir. Nuestra impresora imprime todo lo que recibe del sistema, pero se debe tener cuidado, ya que si no se le envían los datos de una forma ordenada, la impresora puede mezclar las partes de un trabajo con las de otro, incluso dentro del mismo folio. El servicio de impresión, es el encargado de ir enviando los datos de forma correcta a la impresora para que el resultado sea el esperado. Además, las impresoras, no siempre tienen suficiente memoria para guardar todos los datos de impresión de un trabajo completo, por lo que el servicio de impresión se los dará conforme vaya necesitándolos. Cuando finalice cada trabajo, puede notificárselo al usuario. Si en la cola de impresión, no hay trabajos pendientes, el servicio de impresión quedará a la espera y podrá avisar a la impresora para que quede en **StandBy**.

Como este, hay muchos servicios activos o en ejecución en el sistema, y no todos son servicios del sistema operativo, también hay servicios de aplicación, instalados por el usuario y que pueden lanzarse al arrancar el sistema operativo o no, dependiendo de su configuración o cómo los configuremos.

**Un servicio, es un proceso que queda a la espera de que otros le pida que realice una tarea.**

## 2.6. Procesos en Java

### 2.6.1. Creación de procesos

La clase que representa un proceso en Java es la clase ***Process***. Los métodos de ***ProcessBuilder.start() y Runtime.exec()*** crean un proceso nativo en el sistema operativo subyacente donde se está ejecutando la JVM y devuelven una objeto Java de la clase Process que puede ser utilizado para controlar dicho proceso.

- ***Process ProcessBuilder.start():*** inicia un nuevo proceso utilizando los atributos indicados en el objeto. El nuevo proceso ejecuta el comando y los argumentos indicados en el método command(), ejecutándose en el directorio de trabajo especificado por directory(), utilizando las variables de entono definidas en environment().

- ***Process Runtime.exec(String[] cmdarray, String[] envp, File dir):*** ejecuta el comando especificado y argumentos en cmdarray en un proceso hijo independiente con el entorno envp y el directorio de trabajo especificado en dir

Ambos métodos comprueban que el comando a ejecutar es un comando o ejecutable válido en el sistema operativo subyacente sobre el que ejecuta la JVM. El ejecutable se ha podido obtener mediante la compilación de código en cualquier lenguaje de programación. Al final, crear un nuevo proceso depende del sistema operativo en concreto que esté ejecutando por debajo de la JVM. En este sentido, pueden ocurrir múltiples problemas, como:

- No encuentra el ejecutable debido a la ruta indicada.
- No tener permisos de ejecución.
- No ser un ejecutable válido en el sistema.
- etc.

En la mayoría de los casos, se lanza una excepción dependiente del sistema en concreto, pero siempre será una subclase de IOException.

### 2.6.2. Terminación de procesos

Un proceso puede terminar de forma abrupta un proceso hijo que creó. Para ello el proceso padre puede ejecutar la operación ***destroy***. Esta operación elimina el proceso hijo indicado liberando sus recursos en el sistema operativo subyacente. En caso de Java, los recursos correspondientes los eliminará el garbage collector cuando considere.

Si no se fuerza la finalización de la ejecución del proceso hijo de forma anómala, el proceso hijo realizará su ejecución completa terminando y liberando sus recursos al finalizar. Esto se produce cuando el hijo realiza la operación exit para finalizar su ejecución.

### 2.6.3. Comunicación de procesos

Es importante recordar que un proceso es un programa en ejecución y, como cualquier programa, recibe información, la transforma y produce resultados. Esta acción se gestiona a través de:

- La ***entrada estándar (stdin)***: lugar de donde el proceso lee los datos de entrada que requiere para su ejecución. No se refiere a los parámetros de ejecución del programa. Normalmente suele ser el teclado, pero podría recibirlos de un fichero, de la tarjeta de red o hasta de otro proceso, entre otros sitios. La lectura de datos a lo largo de un programa (por ejemplo mediante scanf en C) leerá los datos de su entrada estándar.
- La ***salida estándar (stdout)***: sitio donde el proceso escribe los resultados que obtiene. Normalmente es la pantalla, aunque podría ser, entre otros, la impresora o hasta otro proceso que necesite esos datos como entrada. La escritura de datos que se realice en un programa (por ejemplo mediante printf en C o System.out.println en Java) se produce por la salida estándar.

- La ***salida de error (stderr)***: sitio donde el proceso envía los mensajes de error. Habitualmente es el mismo que la salida estándar, pero puede especificarse que sea otro lugar, por ejemplo un fichero para identificar más fácilmente los errores que ocurren durante la ejecución.

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

En la mayoría de los sistemas operativos, estas entradas y salidas en proceso hijo son una copia de las mismas entradas y salidas que tuviera su padre. De tal forma que si se llama a la operación create dentro de un proceso que lee de un fichero y muestra la salida estándar por pantalla, su hijo correspondiente leerá del mismo fichero y escribirá en pantalla. En Java, en cambio, el proceso hijo creado de la clase Process no tiene su propia interfaz de comunicación, por lo que el usuario no puede comunicarse con él directamente. Todas sus salidas y entradas de información (stdin, stdout y stderr) se redirigen al proceso padre a través de los siguientes *flujos de datos o streams*:

- ***OutputStream***: flujo de salida del proceso hijo. El stream está conectado por un pipe a la entrada estándar (stdin) del proceso hijo.
- ***InputStream***: flujo de entrada del proceso hijo. El stream está conectado por un pipe a la salida estándar (stdout) del proceso hijo.
- ***ErrorStream***: flujo de error del proceso hijo. El stream está conectado por un pipe a la salida estándar (stderr) del proceso hijo. Sin embargo, hay que saber que, por defecto, para la JVM, stderr está conectado al mismo sitio que stdout.

  Si se desea tenerlos separados, lo que permite identificar errores de forma más sencilla, se puede utilizar el método redirectErrorStream(boolean) de la clase ProcessBuilder. Si se pasa un valor true como parámetro, los flujos de datos correspondientes a stderr y stdout en la JVM serán diferentes y representarán la salida estándar y la salida de error del proceso de forma correspondiente.

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

Hay que tener en cuenta que en algunos sistemas operativos, el tamaño de los buffers de entrada y salida que corresponde a stdin y stdout está limitado. En este sentido, un fallo al leer o escribir en los flujos de entrada o salida del proceso hijo puede provocar que el proceso hijo se bloquee. Por eso, en Java se suele realizar la comunicación padre-hijo a través de un buffer utilizando los streams vistos.

### 2.6.4. Sincronización de procesos

Los métodos de comunicación de procesos se pueden considerar como métodos de sincronización ya que permiten al proceso padre llevar el ritmo de envío y recepción de mensajes.

Concretamente, los envíos y recepciones por los flujos de datos permiten a un proceso hijo comunicarse con su padre a través de un canal de comunicación unidireccional bloqueante.

Además de la utilización de los flujos de datos se puede esperar por la finalización del proceso hijo y obtener su información de finalización mediante la ***operación wait***. Dicha operación bloquea al proceso padre hasta que el hijo finaliza su ejecución mediante ***exit***. Como resultado se recibe la información de finalización del proceso hijo. Dicho valor de retorno se especifica mediante un número entero. El valor de retorno significa cómo resulto la ejecución. No tiene nada que ver con los mensajes que se pasan padre e hijo a través de los streams. Por convención se utiliza 0 para indicar que el hijo ha acabado de forma correcta.

Mediante ***waitFor() de la clase Process*** el padre espera bloqueado hasta que el hijo finalice su ejecución, volviendo inmediatamente si el hijo ha finalizado con anterioridad o si alguien le interrumpe (en este caso se lanza la interrupción InterruptedException). Además se puede utilizar ***exitValue()*** para obtener el valor de retorno que devolvió un proceso hijo. El proceso hijo debe haber finalizado, si no, se lanza la excepción IllegalThreadStateException

### 2.6.5. Ejemplos (En todos los casos se usa exec para crear los procesos hijo)

***Ejemplo 1:*** Creación de un proceso hijo para ejecutar el programa *Notepad* y espera del padre a que finalice el hijo

```
import java.io.IOException;
public class Ejemplo1 {


public static void main(String[] args) {

Runtime runtime = Runtime.getRuntime();
String comando = "NOTEPAD";
try {

Process process = runtime.exec(comando);
process.waitFor();

}catch(Exception ex){
System.err.println("Excepción de E/S!!");
System.exit(-1);

}
}
```

```
}
```

***Ejemplo 1 parametrizado:*** Creación de un proceso hijo para ejecutar un programa cuyo nombre se recibe como argumento de entrada y espera del padre a que finalice el hijo

```
import java.io.IOException;
public class Ejemplo1P {


public static void main(String[] args) {
if (args.length <= 0) {
      System.err.println("Se necesita un programa a ejecutar");
      System.exit(-1);
}
Runtime runtime = Runtime.getRuntime();

try {
Process process = runtime.exec(args);

process.waitFor();

}catch(Exception ex){
System.err.println("Excepción de E/S!!");
System.exit(-1);

}
}

}
```

***Ejemplo 2:*** Creación de un proceso hijo para ejecutar el comando *dir* y comunicación de la salida del hijo al padre a través de ***InputStream*** para que el padre lo muestre por pantalla. Además espera del padre a que finalice el hijo y comprobación del valor de retorno que devolvió el proceso hijo

```
import java.io.BufferedReader;
import java.io.InputStream;
import java.io.InputStreamReader;
public class Ejemplo2 {

public static void main(String[] args) {

Runtime runtime = Runtime.getRuntime();
Process process=null;
String comando= "cmd /c dir"; // Para que se ejecute un commando se
invoca al intérprete de comandos cmd
try {
process = runtime.exec(comando);
InputStream is = process.getInputStream();
BufferedReader br = new BufferedReader(new InputStreamReader (is));
String linea;
while ((linea=br.readLine())!=null)
      System.out.println(linea);
br.close();
}catch(Exception ex){
System.err.println("Excepción de E/S!!");
System.exit(-1);
}
```

```
int exitVal;
try {
      exitVal = process.waitFor();
      System.out.println("Valor de salida: "+ exitVal);
}catch(InterruptedException ex) {
      ex.printStackTrace();
}

}
}
```

***Ejemplo 2 parametrizado y con comprobación de errores del proceos hijo:***
Creación de un proceso hijo para ejecutar el comando o programa que el padre reciba como argumento y comunicación de la salida del hijo al padre a través de InputStream para que el padre lo muestre por pantalla. Además espera del padre a que finalice el hijo y comprobación del valor de retorno que devolvió el proceso hijo. Además se usa ***getErrorStream*** para poder leer los posibles errores que se produzcan al lanzar el proceso hijo

```
import java.io.BufferedReader;
import java.io.InputStream;
import java.io.InputStreamReader;
public class Ejemplo2PE {

public static void main(String[] args) {
if (args.length <= 0) {
System.err.println("Se necesita un programa a ejecutar");
System.exit(-1);
}
Runtime runtime = Runtime.getRuntime();
Process process=null;
try {
process = runtime.exec(args);
InputStream is = process.getInputStream();
BufferedReader br = new BufferedReader(new InputStreamReader (is));
String linea;
while ((linea=br.readLine())!=null)
      System.out.println(linea);
br.close();
}catch(Exception ex){
System.err.println("Excepción de E/S!!");
System.exit(-1);
}
int exitVal;
try {
      exitVal = process.waitFor();
      System.out.println("Valor de salida: "+ exitVal);
}catch(InterruptedException ex) {
      ex.printStackTrace();
}
try {
      InputStream er = process.getErrorStream();
      BufferedReader    brer    =    new    BufferedReader    (new
InputStreamReader(er));
      String liner = null;
      while (liner= brer.readLine()) != null)
            System.out.println("ERROR >" + liner);
}catch (IOException ios) {
```

```
      ios.printStackTrace();
}
}
}
```

***Ejemplo 3:*** Creación de un proceso hijo para ejecutar otro programa Java llamado UnSaludo que escribe en un fichero 5 veces un texto recibido por línea de comandos. El proceso padre recibe por línea de comandos el nombre del fichero donde escribirá y recibirá por ***InputStream*** la salida del hijo para que el padre lo escriba en el fichero. Además espera del padre a que finalice el hijo y comprobación del valor de retorno que devolvió el proceso hijo

```
public class UnSaludo {

public static void main(String[] args) {
if (args.length < 1) {
System.err.println("Se necesita un saludo");
System.exit(-1);
}
for (int i=0; i<5; i++)
      System.out.println (i+1 +" ." + args[0]);
}
}
```

```
import java.io.*;

public class Ejemplo3 {

public static void main(String[] args) {

Runtime r = Runtime.getRuntime();
Process p = null;
String comando= "java –jar UnSaludo.jar \"Hola\" "; // Añadir la ruta
donde esté guardado Unsaludo.jar, creado previamente

if (args.length < 1) {
      System.out.println("SE NECESITA UN NOMBRE DE FICHERO ...");
      System.exit(1);
}

try {
// Fichero al que se enviará la salida del programa UnSaludo
FileOutputStream fos = new FileOutputStream (args[0]);
PrintWriter pw = new PrintWriter (fos);

p = r.exec(comando);

InputStream is = p.getInputStream();
BufferedReader br = new BufferedReader(new InputStreamReader (is));
String linea;
while ((linea=br.readLine())!=null){ // Lee la salida de UnSaludo
      System.out.println("INSERTO EN " + args[0] + " > " + linea);
      pw.println(linea); // La inserta en el fichero
}
br.close();
```

```
pw.close();

}catch(Exception ex){
System.err.println("Excepción de E/S!!");
System.exit(-1);
}
int exitVal;
try {
      exitVal = p.waitFor();
      System.out.println("Valor de salida: "+ exitVal);
}catch(InterruptedException ex) {
      ex.printStackTrace();
}

}
}
```

***Ejemplo 4:*** Creación de un proceso hijo para ejecutar el comando *date* y paso de datos del proceso padre al hijo a través de ***OutputStream*** para que el hijo los use para la ejecución de date. Además espera del padre a que finalice el hijo y comprobación del valor de retorno que devolvió el proceso hijo

```
import java.io.*;

public class Ejemplo4 {

public static void main(String[] args) {

Runtime r = Runtime.getRuntime();
Process p = null;
String comando= "cmd /c date ";


try {
p = r.exec(comando);

// Escritura -- envía entrada a DATE
OutputStream os = p.getOutputStream();
os.write ("05-11-2013".getBytes());
os.flush(); //Vacia el buffer

// Lectura -- obtiene la salida de DATE
InputStream is = p.getInputStream();
BufferedReader br = new BufferedReader(new InputStreamReader (is));
String linea;
while ((linea=br.readLine())!=null)
      System.out.println(linea);

br.close();


}

catch(Exception ex){
      System.err.println("Excepción de E/S!!");
      System.exit(-1);
}

int exitVal;
```

```
try {
      exitVal = p.waitFor();
      System.out.println("Valor de salida: "+ exitVal);
}

catch(InterruptedException ex) {
      ex.printStackTrace();
}

}
}
```

## 2.7. Comandos para la gestión de procesos

Es cierto que podemos pensar que ya no necesitamos comandos. Y que podemos desterrar el intérprete de comandos, terminal o shell. Hay múltiples motivos por los que esto no es así:

- En el apartado anterior, hemos visto que necesitamos comandos para lanzar procesos en el sistema. Además de las llamadas al sistema, los comandos son una forma directa de pedirle al sistema operativo que realice tareas por nosotros.
- Construir correctamente los comandos, nos permitirá comunicarnos con el sistema operativo y poder utilizar los resultados de estos comandos en nuestras aplicaciones.
- En GNU/Linux, existen programas en modo texto para realizar casi cualquier cosa. En muchos casos, cuando utilizamos una interfaz gráfica, ésta es un frontend del programa en modo comando. Este frontend, puede proporcionar todas o algunas de las funcionalidades de la herramienta real.
- La administración de sistemas, y más si se realiza de forma remota, es más eficiente en modo comando. Las administradoras y administradores de sistemas experimentadas utilizan scripts y modo comandos, tanto en sistemas Windows como GNU/Linux.

El comienzo en el mundo de los comandos, puede resultar aterrador, hay muchísimos comandos, ¡es imposible aprendérselos todos! Bueno, no nos alarmemos, con este par de trucos podremos defendernos:

- El **nombre de los comandos** suele estar relacionado con la tarea que realizan, sólo que expresado en inglés, o utilizando siglas. Por ejemplo: tasklist muestra un listado de los procesos en sistemas Windows; y en GNU/Linux obtendremos el listado de los procesos con ps, que son las siglas de 'process status'.

- Su **sintaxis** siempre tiene la misma forma:

  **nombreDelComandoopciones**

  Las opciones, dependen del comando en si. Podemos consultar el manual del

comando antes de utilizarlo. En GNU/Linux, lo podemos hacer con "**man** nombreDelComando"; y en Windows, con "nombreDelComando /?"

**Recuerda dejar siempre un espacio en blanco después** del nombreDelComando y entre las opciones.

Después de esos pequeños apuntes, los **comandos que nos interesa conocer para la gestión de procesos** son:

**Windows.** Este sistema operativo es conocido por sus interfaces gráficas, el intérprete de comandos conocido como Símbolo del sistema, no ofrece muchos comandos para la gestión de procesos. Tendremos:

- **tasklist.** Lista los procesos presentes en el sistema. Mostrará el nombre del ejecutable; su correspondiente Identificador de proceso; y, el porcentaje de uso de memoria; entre otros datos.

- **taskkill.** Mata procesos. Con la opción /PID especificaremos el Identificador del proceso que queremos matar.

**GNU/Linux.** En este sistema operativo, todo se puede realizar cualquier tarea en modo texto, además de que los desarrolladores y desarrolladoras respetan en la implementación de las aplicaciones, que sus configuraciones se guarden en archivos de texto plano. Esto es muy útil para las administradoras y administradores de sistemas.

- **ps**. Lista los procesos presentes en el sistema. Con la opción "aux" muestra todos los procesos del sistema independientemente del usuario que los haya lanzado.
- **pstree**. Muestra un listado de procesos en forma de árbol, mostrando qué procesos han creado otros. Con la opción "AGu" construirá el árbol utilizando líneas guía y mostrará el nombre de usuario propietario del proceso.
- **kill**. Manda señales a los procesos. La señal -9, matará al proceso. Se utiliza "kill -9 <PID>". killall. Mata procesos por su nombre. Se utiliza como "killall nombreDeAplicacion".
- **nice**. Cambia la prioridad de un proceso. "nice -n 5 comando" ejecutará el comando con una prioridad 5. Por defecto la prioridad es 0. Las prioridades están entre -20 (más alta) y 19 (más baja).

## 2.8. Herramientas gráficas para la gestión de procesos

Tanto los sistemas Windows como GNU/Linux proporcionan herramientas gráficas para la gestión de procesos. En el caso de Windows, se trata del **Administrador de tareas**, y en GNU/Linux del **Monitor del sistema**. Ambos, son bastante parecidos, nos ofrecen, al menos, las siguientes funcionalidades e información:

- Listado de todos los procesos que se encuentran activos en el sistema,

mostrando su PID, usuario y ubicación de su fichero ejecutable.
- Posibilidad de finalizar procesos.
- Información sobre el uso de CPU, memoria principal y virtual, red, … Posibilidad de cambiar la prioridad de ejecución de los procesos.

# 3. Programación concurrente

Hasta ahora hemos programado aplicaciones secuenciales u orientadas a eventos. Siempre hemos pensado en nuestras aplicaciones como si se ejecutaran de forma aislada en la máquina. De hecho, el SO garantiza que un proceso no accede al espacio de trabajo (zona de memoria) de otro, esto es, unos procesos no pueden acceder a las variables de otros procesos. Sin embargo, los procesos, en ocasiones, necesitan comunicarse entre ellos, o necesitan acceder al mismo recurso (fichero, dispositivo, etc.). En esas situaciones, hay que **controlar la forma en la que esos procesos se comunican o acceden a los recursos, para que no haya errores, resultados incorrectos o inesperados**.

Podemos ver la concurrencia como una carrera, en la que todos los corredores corren al mismo tiempo buscando un mismo fin, que es ganar la carrera. En el caso de los procesos, competirán por conseguir todos los recursos que necesiten.

La definición de **concurrencia**, no es algo sencillo. En el diccionario, **concurrencia es la coincidencia de varios sucesos al mismo tiempo**.

Nosotros podemos decir que **dos procesos son concurrentes**, cuando **la primera instrucción de un proceso se ejecuta después de la primera y antes de la última de otro proceso**.

Por otro lado, hemos visto que los procesos activos se ejecutan alternando sus instantes de ejecución en la CPU. Y, aunque nuestro equipo tenga más de un núcleo, los tiempos de ejecución de cada núcleo se repartirán entre los distintos procesos en ejecución. **La planificación alternando los instantes de ejecución en la gestión de los procesos, hace que los procesos se ejecuten de forma concurrente.** O lo que es lo mismo:

multiproceso **= concurrencia**.

La **programación concurrente** proporciona **mecanismos de comunicación** y **sincronización** entre procesos que se ejecutan de forma simultánea en un sistema informático. La programación concurrente nos permitirá definir qué **instrucciones** de nuestros procesos se pueden ejecutar de forma simultánea con las de otros procesos, sin que se produzcan errores; y cuáles deben ser **sincronizadas** con las de otros procesos **para que los resultados de sean correctos**.

## 3.1. ¿Para qué concurrencia?

Las **principales razones** por las que se utiliza una estructura concurrente son:

- **Optimizar la utilización de los recursos.** Podremos simultanear las operaciones de E/S en los procesos. La CPU estará menos tiempo ociosa. Un equipo informático es como una cadena de producción, obtenemos más productividad realizando las tareas concurrentemente.

- **Proporcionar interactividad a los usuarios (y animación gráfica).** Todos nos hemos desesperado esperando que nuestro equipo finalizara una tarea. Esto se agravaría si no existiera el multiprocesamiento, sólo podríamos ejecutar procesos por lotes.

- **Mejorar la disponibilidad.** Servidor que no realice tareas de forma concurrente, no podrá atender peticiones de clientes simultáneamente.

- **Conseguir un diseño conceptualmente más comprensible y mantenible.** El diseño concurrente de un programa nos llevará a una mayor modularidad y claridad. Se diseña una solución para cada tarea que tenga que realizar la aplicación (no todo mezclado en el mismo algoritmo). Cada proceso se activará cuando sea necesario realizar cada tarea.

- **Aumentar la protección.** Tener cada tarea aislada en un proceso permitirá depurar la seguridad de cada proceso y, poder finalizarlo en caso de mal funcionamiento sin que suponga la caída del sistema.

Los anteriores pueden parecer los motivos para utilizar concurrencia en sistemas con un solo procesador. Los actuales avances tecnológicos hacen necesario **tener en cuenta** la concurrencia en el diseño de las aplicaciones para aprovechar su potencial. Los nuevos entornos hardware son:

- **Microprocesadores con múltiples** núcleos que comparten la memoria principal del sistema.

- **Entornos multiprocesador con memoria compartida.** Todos los procesadores utilizan un mismo espacio de direcciones a memoria, sin tener conciencia de dónde están instalados físicamente los módulos de memoria.

- **Entornos distribuidos.** Conjunto de equipos heterogéneos o no, conectados por red y/o Internet.

Los **beneficios** que obtendremos al adoptar un modelo de programa concurrente son:

- Estructurar un programa como conjunto de procesos concurrentes que interactúan, **aporta gran claridad** sobre lo que cada proceso debe hacer y cuando debe hacerlo.

- **Puede conducir a una reducción del tiempo de ejecución.** Cuando se trata de un entorno monoprocesador, permite solapar los tiempos de E/S o

de acceso al disco de unos procesos con los tiempos de ejecución de CPU de otros procesos. Cuando el entorno es multiprocesador, la ejecución de los procesos es realmente simultánea en el tiempo (paralela), y esto reduce el tiempo de ejecución del programa.

- **Permite una mayor flexibilidad de planificación.** Procesos de alta prioridad pueden ser ejecutados antes de otros procesos menos urgentes.

- La concepción concurrente del software **permite un mejor modelado** previo **del comportamiento del programa**, y en consecuencia un **análisis más fiable** de las diferentes opciones que requiera su diseño.

## 3.2. Condiciones de competencia

Acabamos de ver que tenemos que desechar la idea de que nuestra aplicación se ejecutará de forma aislada. Y que, de una forma u otra, va a interactuar con otros procesos. Distinguimos los siguientes **tipos básicos de interacción entre procesos concurrentes**:

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

En el segundo y tercer caso, **necesitamos componentes que nos permitan establecer acciones de sincronización y comunicación entre los procesos**.

Un proceso entra en **condición de competencia** con otro, cuando **ambos necesitan el mismo recurso, ya sea forma exclusiva o no;** por lo que será necesario utilizar mecanismos de sincronización y comunicación entre ellos.

Un ejemplo sencillo de **procesos cooperantes**, es "un proceso recolector y un proceso productor". El proceso recolector necesita la información que el otro proceso produce. El proceso **recolector**, quedará bloqueado mientras que no haya información disponible.

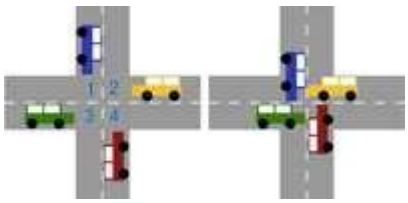
El proceso **productor**, puede escribir siempre que lo desee (es el único que produce ese tipo de información). Por supuesto, podemos complicar esto, con varios procesos recolectores para un sólo productor; y si ese productor puede dar información a todos los recolectores de forma simultánea o no; o a cuántos procesos recolectores puede dar servicio de forma concurrente. Para determinar si los recolectores tendrán que esperar su turno o no. Pero ya abordaremos las soluciones a estas situaciones más adelante.

En el caso de procesos competidores, vamos a comenzar viendo unas **definiciones**:

- Cuando un proceso necesita un recurso de forma exclusiva, es porque mientras que lo esté utilizando él, ningún otro puede utilizarlo. Se llama región de exclusión mutua o región crítica al **conjunto de instrucciones en las que el proceso utiliza un recurso y que se deben ejecutar de forma exclusiva con respecto a otros procesos competidores por ese mismo recurso.**

- Cuando más de un proceso necesitan el mismo recurso, antes de utilizarlo tienen que pedir su uso, una vez que lo obtienen, el resto de procesos quedarán bloqueados al pedir ese mismo recurso. Se dice que un proceso hace **un lock (bloqueo) sobre un recurso cuando ha obtenido su uso en exclusión mutua.**
- Por ejemplo dos procesos, compiten por dos recursos distintos, y ambos necesitan ambos recursos para continuar. Se puede dar la situación en la que cada uno de los procesos bloquee uno de los recursos, lo que hará que el otro proceso no pueda obtener el recurso que le falta; quedando bloqueados un proceso por el otro sin poder finalizar. **Deadlock o** interbloqueo**, se produce cuando los procesos no pueden obtener, nunca, los recursos necesarios para continuar su tarea.** El interbloqueo es una situación muy peligrosa, ya que puede llevar al sistema a su caída o cuelgue.

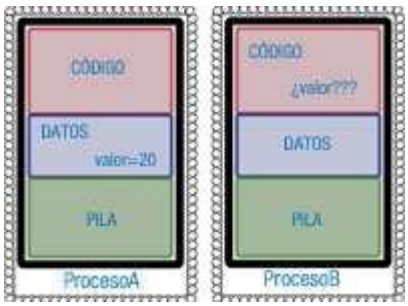
¿Crees que no es usual que pueda darse una situación de interbloqueo? Veamos un ejemplo sencillo: un cruce de caminos y cuatro coches.



El coche azul necesita las regiones 1 y 3 para continuar, el amarillo: 2 y 1, el rojo: 4 y 2, y el verde: 3 y 4.

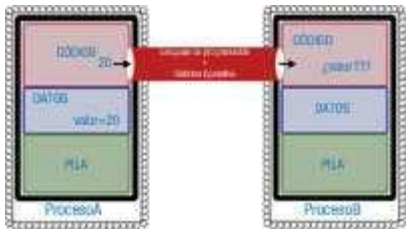
Obviamente, no siempre quedarán bloqueados, pero se puede dar la situación en la que ninguno ceda. Entonces, quedarán interbloqueados.

# 4. Comunicación entre procesos




Como ya hemos comentado en más de una ocasión a lo largo de esta unidad, cada proceso tiene su espacio de direcciones privado, al que no pueden acceder el resto de procesos. Esto constituye un mecanismo de seguridad; imagina qué locura, si tienes un dato en tu programa y cualquier otro, puede modificarlo de cualquier manera. Tu programa generaría errores, como poco.

Por supuesto, nos damos cuenta de que, si cada proceso tiene sus datos y otros procesos no pueden acceder a ellos directamente, cuando otro proceso los necesite, tendrá que existir alguna forma de comunicación entre ellos.
**Comunicación entre procesos: un proceso da o deja información; recibe o recoge información.**



Los **lenguajes de programación y los sistemas operativos, nos proporcionan** primitivas **de sincronización que facilitan la interacción entre procesos de forma sencilla y eficiente.**

Una primitiva, hace referencia a una operación de la cual conocemos sus restricciones y efectos, pero no su implementación exacta. Veremos que usar esas primitivas se traduce en utilizar **objetos y** sus **métodos**, teniendo muy en cuenta sus **repercusiones reales en el comportamiento de nuestros procesos.**
Clasificaremos las interacciones entre los procesos y el resto del sistema (recursos y otros procesos), como estas tres:

- **Sincronización:** Un proceso puede conocer el punto de ejecución en el que se encuentra otro en ese determinado instante.
- **Exclusión mutua:** Mientras que un proceso accede a un recurso, ningún otro proceso accede al mismo recurso o variable compartida.
- **Sincronización condicional:** Sólo se accede a un recurso cuando se encuentra en un determinado estado interno.

## 4.1. Mecanismos básicos de comunicación

Si pensamos en la forma en la que un proceso puede comunicarse con otro. Se nos ocurrirán estas dos:

- **Intercambio de mensajes.** Tendremos las primitivas enviar (**send**) y recibir (**receive** o **wait**) información.

- **Recursos (o memoria) compartidos.** Las primitivas serán escribir (write) y leer (read) datos en o de un recurso.

En el caso de **comunicar procesos dentro de una misma máquina**, el **intercambio de mensajes**, se puede realizar de dos formas:

- Utilizar un **buffer de memoria**.

- Utilizar un **socket**.

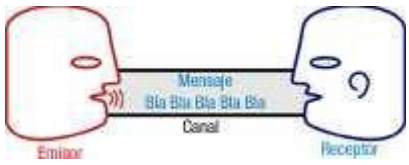
La diferencia entre ambos, está en que un socket se utiliza para intercambiar información entre procesos en distintas máquinas a través de la red; y un buffer de memoria, crea un canal de comunicación entre dos procesos utilizando la memoria principal del sistema. Actualmente, es más común el uso de sockets que buffers para comunicar procesos. Trataremos en profundidad los sockets en posteriores unidades. Pero veremos un par de ejemplos muy sencillos de ambos.

En java, **utilizaremos sockets y buffers como** si utilizáramos **cualquier otro stream o flujo de datos**. Utilizaremos **los métodos read-write** en lugar de send-receive.

Con respecto a las **lecturas y escrituras**, debemos recordar, que serán **bloqueantes**. Es decir, un proceso quedará bloqueado hasta que los datos estén listos para poder ser leídos. Una escritura, bloqueará al proceso que intenta escribir, hasta que el recurso no esté preparado para poder escribir. Aunque, esto está relacionado con el acceso a recursos compartidos, cosa que estudiaremos en profundidad, en el apartado 5.1 Regiones críticas.

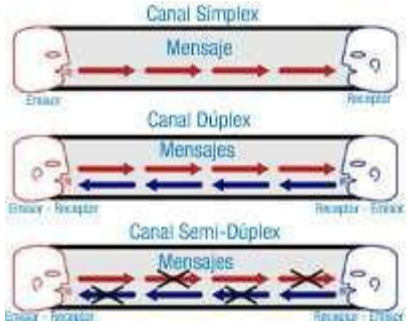
Pero, esto parece que va a ser algo complicado. ¿Cómo implementamos la comunicación entre procesos?

## 4.2. Tipos de comunicación




Ya hemos visto que dos procesos pueden comunicarse. Remarquemos algunos conceptos fundamentales sobre comunicación.
En cualquier comunicación, vamos a tener los siguientes **elementos**: **Mensaje**. Información que es el objeto de la comunicación.
- **Emisor**. Entidad que emite, genera o es origen del mensaje.
- **Receptor**. Entidad que recibe, recoge o es destinataria del mensaje.
- **Canal**. Medio por el que viaja o es enviado y recibido el mensaje.

Podemos clasificar el **canal de comunicación** según su capacidad, y los sentidos en los que puede viajar la información, como:




- **Símplex**. La comunicación se produce en un sólo sentido. El emisor es origen del mensaje y el receptor escucha el mensaje al final del canal. Ejemplo: reproducción de una película en una sala de cine.
- **Dúplex** (Full Duplex). Pueden viajar mensajes en ambos sentidos simultáneamente entre emisor y receptor. El emisor es también receptor y el receptor es también emisor. Ejemplo: telefonía.
- **Semidúplex** (Half Duplex). El mensaje puede viajar en ambos sentidos, pero no al mismo tiempo. Ejemplo: comunicación con walkie-talkies.

Otra clasificación dependiendo de la **sincronía** que mantengan el **emisor y** el **receptor** durante la comunicación, será:

- **Síncrona**. El emisor queda bloqueado hasta que el receptor recibe el mensaje. Ambos se sincronizan en el momento de la recepción del mensaje.
- **Asíncrona**. El emisor continúa con su ejecución inmediatamente después de emitir el mensaje, sin quedar bloqueado.
- **Invocación remota**. El proceso emisor queda suspendido hasta que recibe la confirmación de que el receptor recibido correctamente el mensaje, después emisor y receptor ejecutarán síncronamente un segmento de código común.

Dependiendo del **comportamiento** que tengan los **interlocutores que intervienen en la comunicación**, tendremos comunicación:

- **Simétrica**. Todos los procesos pueden enviar y recibir información.
- **Asimétrica**. Sólo un proceso actúa de emisor, el resto sólo escucharán el o los mensajes.

En nuestro anterior ejemplo básico de comunicación con sockets: el proceso **SocketEscritor**, era el emisor; el proceso **SocketLector**, era el receptor. El canal de comunicación: sockets. En el ejemplo, hemos utilizado del socket en una sola dirección y síncrona; pero los sockets permiten comunicación dúplex síncrona (en cada sentido de la comunicación) y simétrica (ambos procesos pueden escribir en y leer del socket); también existen otros tipos de sockets que nos permitirán establecer comunicaciones asimétricas asíncronas (**DatagramSocket**).

En el caso del ejemplo de las tuberías, la comunicación que se establece es simplex síncrona y asimétrica.

Nos damos cuenta, que conocer las características de la comunicación que necesitamos establecer entre procesos, nos permitirá seleccionar el canal, herramientas y comportamiento más convenientes.

# 5. Sincronización entre procesos

Ya tenemos mucho más claro, que las situaciones en las que dos o más procesos tengan que comunicarse, cooperar o utilizar un mismo recurso; implicará que deba haber cierto **sincronismo** entre ellos. O bien, unos tienen que esperar que otros finalicen alguna acción; o, tienen que realizar alguna tarea al mismo tiempo.

En este capítulo, veremos distintas **problemáticas, primitivas y soluciones de sincronización** necesarias para resolverlas. También es cierto, que en **el sincronismo entre procesos lo hace posible el SO, y lo que hacen los lenguajes de programación de alto nivel es encapsular los mecanismos de sincronismo que proporciona cada SO en objetos, métodos y funciones.** Los lenguajes de programación, proporcionan primitivas de sincronismo entre los distintos hilos que tenga un proceso; estas primitivas del lenguaje, las veremos en la siguiente unidad.

Comencemos viendo un ejemplo muy sencillo de un problema que se nos plantea de forma más o menos común: inconsistencias en la actualización de un valor compartido por varios procesos; así, nos daremos cuenta de la importancia del uso de mecanismos de sincronización.

En programación concurrente, siempre que accedamos a algún **recurso compartido** (eso incluye a los **ficheros**), deberemos tener en cuenta las **condiciones** en las que **nuestro proceso debe hacer uso de ese recurso**: ¿será de forma exclusiva o no? Lo que ya definimos anteriormente como **condiciones de competencia**.

En el caso de lecturas y escrituras en un fichero, debemos determinar si queremos acceder al fichero como sólo lectura; escritura; o lectura-escritura; y utilizar los objetos que nos permitan establecer los mecanismos de sincronización necesarios para que un proceso pueda bloquear el uso del fichero por otros procesos cuando él lo esté utilizando.

Esto se conoce como el **problema de los procesos lectores-escritores**. El **sistema operativo, nos ayudará** a resolver los problemas que se plantean; ya que:

- Si el acceso es de **sólo lectura**. Permitirá que todos los procesos lectores, que sólo quieren leer información del fichero, **puedan acceder simultáneamente** a él.

- En el caso de **escritura**, o lectura-escritura. El SO nos permitirá pedir un tipo

de **acceso** de **forma exclusiva** al fichero. Esto significará que el proceso deberá esperar a que otros procesos lectores terminen sus accesos. Y otros procesos (lectores o escritores), esperarán a que ese proceso escritor haya finalizado su escritura.

Debemos tener en cuenta que, nosotros, **nos comunicamos con el SO a través de los objetos y métodos proporcionados por un lenguaje de programación**; y, por lo tanto, tendremos que **consultar cuidadosamente la documentación de las clases** que estamos utilizando para **conocer** todas las **peculiaridades** de su comportamiento.

## 5.1. Regiones críticas

La definición común, y que habíamos visto anteriormente, de una región o sección crítica, es, el **conjunto de instrucciones en las que un proceso accede a un recurso compartido**. Para que la definición sea correcta, añadiremos que, las instrucciones que forman esa región crítica, **se ejecutarán de forma indivisible o atómica y de forma exclusiva con respecto a otros procesos que accedan al mismo recurso** compartido al que se está accediendo.

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

**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.

Debemos darnos cuenta de que nos referimos a un recurso compartido de forma genérica, ese recurso compartido podrá ser: memoria principal, fichero, base de datos, etc.

## 5.2. Semáforos

Veamos una primera solución eficiente a los problemas de sincronismo, entre procesos que acceden a un mismo recurso compartido.

Podemos ver varios procesos que quieren acceder al mismo recurso, como coches que necesitan pasar por un cruce de calles. En nuestros cruces, los semáforos nos indican cuándo podemos pasar y cuándo no. Nosotros, antes de intentar entrar en el cruce, primero miramos el color en el que se encuentra el semáforo, y si está en verde (abierto), pasamos. Si el color es rojo (cerrado), quedamos a la espera de que ese mismo semáforo nos indique que podemos pasar. Este mismo funcionamiento es el que van a seguir nuestros semáforos en programación concurrente. Y, son una solución eficiente, porque los procesos quedarán bloqueados (y no en espera activa) cuando no puedan acceder al recurso, y será el semáforo el que vaya desbloqueándolos cuando puedan pasar.

Un **semáforo**, es un componente de bajo nivel de abstracción que permite arbitrar los accesos a un recurso compartido en un entorno de programación concurrente.

Al **utilizar** un semáforo, lo veremos como un **tipo dato**, que podremos instanciar. Ese objeto semáforo podrá tomar un determinado **conjunto de valores y** se podrá realizar con él un **conjunto** determinado **de operaciones**. Un semáforo, **tendrá también asociada una lista de procesos que suspendidos que se encuentran a la espera de entrar en el mismo**.

Dependiendo del conjunto de datos que pueda tomar un semáforo, tendremos:

- Semáforos **binarios**. Aquellos que pueden tomar sólo valores 0 ó 1. Como nuestras luces verde y roja.
- Semáforos **generales**. Pueden tomar cualquier valor Natural (entero no negativo).

En cualquier caso, los valores que toma un semáforo representan: Valor igual a 0. Indica que el semáforo está cerrado. Valor mayor de 0. El semáforo está abierto.

Cualquier **semáforo** permite **dos operaciones seguras** (la implementación del semáforo garantiza que la operación de chequeo del valor del semáforo, y posterior actualización según proceda, es siempre segura respecto a otros accesos concurrentes):

- objSemaforo.wait(): Si el semáforo no es nulo (está abierto) decrementa en uno el valor del semáforo. Si el valor del semáforo es nulo (está cerrado), el proceso

que lo ejecuta se suspende y se encola en la lista de procesos en espera del semáforo.

- objSemaforo.signal(): Si hay algún proceso en la lista de procesos del semáforo, activa uno de ellos para que ejecute la sentencia que sigue al wait que lo suspendió. Si no hay procesos en espera en la lista incrementa en 1 el valor del semáforo.

Además de la operación segura anterior, con un semáforo, también podremos realizar una operación no segura, que es la **inicialización del valor del semáforo**. Ese valor indicará cuántos procesos pueden entrar concurrentemente en él. Esta inicialización la realizaremos al crear el semáforo.

Para utilizar semáforos, seguiremos los siguientes pasos:

1.- Un proceso padre creará e inicializará tanto semáforo.
2.- El proceso padre creará el resto de procesos hijo pasándoles el semáforo que ha creado. Esos procesos hijos acceden al mismo recurso compartido.
3.- Cada proceso hijo, hará uso de las operaciones seguras wait y signal respetando este esquema:

3.1.- objSemaforo.wait(); Para consultar si puede acceder a la sección crítica.
3.2.- Sección crítica; Instrucciones que acceden al recurso protegido por el semáforo objSemaforo.
3.3.- objSemaforo.signal(); Indicar que abandona su sección y otro proceso podrá entrar.
3.4.- El proceso padre habrá creado tantos semáforos como tipos secciones críticas distintas se puedan distinguir en el funcionamiento de los procesos hijos (puede ocurrir, uno por cada recurso compartido).

La ventaja de utilizar semáforos es que son fáciles de comprender, proporcionan una **gran capacidad funcional (podemos utilizarlos para resolver cualquier problema de concurrencia)**. Pero, su nivel bajo de abstracción, los hace **peligrosos de manejar** y, a menudo, son la **causa de muchos errores**, como es el interbloqueo. Un simple olvido o cambio de orden conduce a bloqueos; y requieren que la gestión de un semáforo se distribuya por todo el código lo que hace la depuración de los errores en su gestión es muy difícil.
En **java**, encontramos la clase **Semaphore** dentro del paquete **java.util.concurrent**; y su uso real **se aplica** a los **hilos** de un mismo proceso, para arbitrar el acceso de esos hilos de forma concurrente a una misma región de la memoria del proceso. Por ello, veremos ejemplos de su uso en siguiente unidades de este módulo.

## 5.3. Monitores

Los monitores, nos ayudan a resolver las desventajas que encontramos en el uso de semáforos. El problema en el uso de semáforos es que, recae sobre el programador o programadora la tarea implementar el correcto uso de cada semáforo para la protección de cada recurso compartido; y, sigue estando disponible el recurso para utilizarlo sin la protección de un semáforo. Los monitores son como guardaespaldas, encargados de la protección de uno o varios recursos específicos; pero encierran esos recursos de forma que el proceso sólo puede acceder a esos recursos a través de los métodos que el monitor expone.

Un **monitor**, es un componente de alto nivel de abstracción destinado a gestionar recursos que van a ser accedidos de forma concurrente.

Los monitores encierran en su interior los recursos o variables compartidas como componentes privadas y garantizan el acceso a ellas en exclusión mutua.
La declaración de un **monitor incluye**:

- Declaración de las **constantes, variables, procedimientos** y **funciones** que son **privados del monitor** (solo el monitor tiene visibilidad sobre ellos).

- Declaración de los **procedimientos** y **funciones** que el monitor expone (**públicos**) y que constituyen la **interfaz a través de las que los procesos acceden al monitor**.

- **Cuerpo del monitor**, constituido por un bloque de código que se ejecuta al ser instanciado o inicializado el monitor. Su finalidad es **inicializar las variables y estructuras internas del monitor**.

El **monitor garantiza el acceso al código interno en régimen de exclusión mutua.**
Tiene asociada una **lista** en la que se incluyen los **procesos que al tratar de acceder al monitor son suspendidos**.

Los paquetes Java, no proporcionan una implementación de clase **Monitor** (habría que implementar un monitor para cada variable o recurso a sincronizar). Pero, siempre **podemos implementarnos nuestra propia clase monitor, haciendo uso de semáforos para ello**.

Pensemos un poco, ¿hemos utilizado objetos que pueden encajar con la declaración de un monitor aunque su tipo de dato no fuera monitor?, ¿no?, ¿seguro? Cuando realizamos una lectura o escritura en fichero, nuestro proceso queda bloqueado hasta que el sistema ha realizado completamente la operación. Nosotros inicializamos el uso del fichero indicando su ruta al crear el objeto, por ejemplo, **FileReader**; y utilizamos los métodos expuestos por ese objeto para realizar las operaciones que deseamos con ese fichero. Sin embargo, el código

que realmente realiza esas operaciones es el implementado en la clase **FileReader**. Si bien, esos objetos no proporcionan exclusión mutua en los accesos al recurso; o, por lo menos, no en todos sus métodos. Aún así, podemos decir que **utilicemos objetos de tipo monitor al acceder a los recursos del sistema**, aunque no tengan como nombre Monitor.

Las **ventajas** que proporciona el uso de monitores son:

- Uniformidad: El monitor provee una única capacidad, la exclusión mutua, no existe la confusión de los semáforos.
- Modularidad: El código que se ejecuta en exclusión mutua está separado, no mezclado con el resto del programa.
- Simplicidad: El programador o programadora no necesita preocuparse de las herramientas para la exclusión mutua .
- Eficiencia de la implementación: La implementación subyacente puede limitarse fácilmente a los semáforos.

Y, la **desventaja**:

- Interacción de múltiples condiciones de sincronización: Cuando el número de condiciones crece, y se hacen complicadas, la complejidad del código crece de manera extraordinaria.

## 5.4. Memoria compartida

Una **forma natural de comunicación entre procesos** es la posibilidad de disponer de zonas de memoria compartidas (variables, buffers o estructuras). Además, los mecanismos de sincronización en programación concurrente que hemos visto: **regiones críticas, semáforos y monitores**; tienen su razón de ser en la existencia de **recursos compartidos**; incluida la memoria compartida.

Cuando se crea un proceso, el sistema operativo le asigna los recursos iniciales que necesita, siendo el principal recurso: la **zona de memoria en la que se guardarán sus instrucciones, datos y pila de ejecución**. Pero como ya hemos comentado anteriormente, los sistemas operativos modernos, implementan **mecanismos que permiten proteger la zona de memoria de cada proceso** siendo ésta **privada** para cada proceso, de forma que otros no podrán acceder a ella. Con esto, podemos pensar que no hay posibilidad de tener comunicación entre procesos por medio de memoria compartida. Pues, no es así. En la actualidad, la **programación multihilo** (que abordaremos en la siguiente unidad y, se refiere, a tener varios flujos de ejecución dentro de un mismo proceso, compartiendo entre ellos la memoria asignada al proceso), nos permitirá examinar al máximo esta funcionalidad.

Pensemos ahora en unos **problemas** que pueden resultar **complicados** si los resolvemos con un sólo procesador, por ejemplo: la ordenación de los elementos

de una matriz. Ordenar una matriz pequeña, no supone mucho problema; pero si la matriz se hace muy grande... Si disponemos de **varios procesadores** y somos capaces de partir la matriz en trozos (**convertir un problema grande en varios más pequeños**) de forma que cada procesador se encargue de ordenar cada parte de la matriz. Conseguiremos resolver el problema en menos tiempo; eso sí, teniendo en cuenta la **complejidad de dividir el problema y asignar a cada procesador el conjunto de datos (o zona de memoria) que tiene que manejar y la tarea o proceso a realizar** (y finalizar con la tarea de **combinar todos los resultados para obtener la solución final**). En este caso, tenemos **sistemas multiprocesador** como los actuales microprocesadores de varios núcleos, o los **supercomputadores** formados por múltiples ordenadores completos (e idénticos) trabajando como un único sistema. En ambos casos, **contaremos con ayuda de sistemas específicos (sistemas operativos o entornos de programación), preparados para soportar la carga de computación en múltiples núcleos y/o equipos.**

## 5.5. Cola de mensajes

El paso de mensajes es una **técnica** empleada en programación concurrente para aportar **sincronización entre procesos y permitir la exclusión mutua**, de manera similar a como se hace con los semáforos, monitores, etc. Su principal característica es que **no precisa de memoria compartida**.

Los **elementos principales** que intervienen en el paso de mensajes son el **proceso** que **envía**, el que **recibe** y el **mensaje**.

Dependiendo de si el proceso que envía el mensaje espera a que el mensaje sea recibido, se puede hablar de paso de mensajes síncrono o asíncrono:

- En el **paso de mensajes asíncrono**, el proceso que envía, **no espera a que el mensaje sea recibido**, y continúa su ejecución, siendo posible que vuelva a generar un nuevo mensaje y a enviarlo antes de que se haya recibido el anterior. Por este motivo se suelen emplear **buzones o colas**, en los que se **almacenan los mensajes a espera de que un proceso los reciba**. Generalmente empleando este sistema, el proceso que envía mensajes sólo se bloquea o para, cuando finaliza su ejecución, o si el buzón está lleno. Para conseguir esto, estableceremos una serie de **reglas de comunicación** (o protocolo) entre emisor y receptor, de forma que el **receptor** pueda **indicar al emisor qué capacidad restante** queda en su cola de mensajes y si está lleno o no.

- En el **paso de mensajes síncrono**, el proceso que envía el mensaje **espera a que un proceso lo reciba** para continuar su ejecución. Por esto se suele llamar a esta técnica encuentro, o rendezvous. Dentro del paso de mensajes síncrono se engloba a la **llamada a procedimiento remoto** (RPC), muy popular en las arquitecturas cliente/servidor.

# 6. Requisitos: seguridad, vivacidad, eficiencia y reusabilidad.

Como cualquier aplicación, los programas concurrentes deben cumplir una serie de requisitos de calidad. En este apartado, veremos algunos aspectos que nos permitirán desarrollar proyectos concurrentes con, casi, la completa certeza de que estamos **desarrollando software de calidad**.

Todo **programa concurrente** debe **satisfacer** dos tipos de **propiedades**:

- Propiedades de **seguridad** ("**safety**"): estas propiedades son relativas a que **en cada instante** de la ejecución **no debe haberse producido algo que haga entrar al programa en un estado erróneo**:
  - Dos procesos **no deben entrar simultáneamente en una sección crítica.**
  - Se **respetan las condiciones de sincronismo**, como: el consumidor no debe consumir el dato antes de que el productor los haya producido; y, el productor no debe producir un dato mientras que el buffer de comunicación esté lleno.
- Propiedades de **vivacidad** ("**liveness**"): cada sentencia que se ejecute conduce en algún modo a **un avance constructivo para alcanzar el objetivo funcional del programa**. Son, en general, muy dependientes de la política de planificación que se utilice. Ejemplos de propiedades de vivacidad son:
  - No deben producirse **bloqueos activos** (**livelock**). Conjuntos de procesos que ejecutan de forma continuada sentencias que no conducen a un progreso constructivo.
  - **Aplazamiento indefinido** (**starvation**): consiste en el estado al que puede llegar un programa que aunque potencialmente puede avanzar de forma constructiva. Esto puede suceder, como consecuencia de que no se le asigna tiempo de procesador en la política de planificación; o, porque en las condiciones de sincronización hemos establecido criterios de prioridad que perjudican siempre al mismo proceso.
  - **Interbloqueo** (**deadlock**): se produce cuando los procesos no pueden obtener, nunca, los recursos necesarios para finalizar su tarea. Vimos un ejemplo de esta situación en el apartado 4.2 Condiciones de competencia.
- Es evidentemente, que también nos preocuparemos por diseñar nuestras aplicaciones para que sean **eficientes:**
  - No utilizarán más **recursos** de los **necesarios**.
  - Buscaremos la **rigurosidad** en su **implementación: toda la funcionalidad esperada de forma correcta y concreta.**

- Y, en cuanto a la **reusabilidad**, debemos tenerlo ya, muy bien aprendido:

  - Implementar el código de forma **modular**: definiendo clases, métodos, funciones, ...
  - **Documentar** correctamente el código y el proyecto.

Para conseguir todos lo anterior contaremos con los **patrones de diseño**; y pondremos especial cuidado en **documentar** y **depurar** convenientemente.

## 6.1. Arquitecturas y patrones de diseño.

La Arquitectura del Software, también denominada arquitectura lógica, es el **diseño de más alto nivel** de la estructura de un sistema. Consiste en un **conjunto de patrones y abstracciones coherentes** con base a las cuales se pueden resolver los problemas. A semejanza de los planos de un edificio o construcción, estas indican la **estructura, funcionamiento e interacción entre las partes del software**.

La arquitectura de software, tiene que ver con el diseño y la implementación de estructuras de software de alto nivel. Es el resultado de ensamblar un cierto número de elementos arquitectónicos de forma adecuada para **satisfacer la mayor funcionalidad y requerimientos de desempeño** de un sistema, así como requerimientos no funcionales, como la **confiabilidad, escalabilidad, portabilidad, y disponibilidad; mantenibilidad, auditabilidad, flexibilidad e interacción**.

Generalmente, no es necesario inventar una nueva arquitectura de software para cada sistema de información. Lo habitual es **adoptar una arquitectura conocida en función de sus ventajas e inconvenientes** para cada caso en concreto. Así, las arquitecturas **más universales** son:

- **Monolítica**. El software se estructura en grupos funcionales muy acoplados.

- **Cliente-servidor**. El software reparte su carga de cómputo en dos partes independientes: consumir un servicio y proporcionar un servicio.

- **Arquitectura de tres niveles**. Especialización de la arquitectura cliente-servidor donde la carga se divide en capas con un reparto claro de funciones: una capa para la presentación (interfaz de usuario), otra para el cálculo (donde se encuentra modelado el negocio) y otra para el almacenamiento (persistencia). Una capa solamente tiene relación con la siguiente.

Otras arquitecturas menos conocidas son:

- En **pipeline**. Consiste en modelar un proceso comprendido por varias fases secuenciales, siendo la entrada de cada una la salida de la anterior.

- **Entre pares**. Similar a la arquitectura cliente-servidor, salvo porque podemos decir que cada elemento es igual a otro (actúa simultáneamente como cliente y como servidor).

- En **pizarra**. Consta de múltiples elementos funcionales (llamados agentes) y un instrumento de control o pizarra. Los agentes estarán especializados en una tarea concreta o elemental. El comportamiento básico de cualquier agente, es: examinar la pizarra, realizar su tarea y escribir sus conclusiones en la misma pizarra. De esta manera, otro agente puede trabajar sobre los resultados generados por otro

- **Orientada a servicios** (Service Oriented Architecture - SOA) Se diseñan servicios de aplicación basados en una definición formal independiente de la plataforma subyacente y del lenguaje de programación, con una interfaz estándar; así, un servicio C# podrá ser usado por una aplicación Java.

- **Dirigida por eventos**. Centrada en la producción, detección, consumo de, y reacción a eventos.

Los patrones de diseño, se definen como soluciones de diseño que son válidas en distintos contextos y que han sido aplicadas con éxito en otras ocasiones:

- Ayudan a "arrancar" en el diseño de un programa complejo.
- Dan una descomposición de objetos inicial "bien pensada". Pensados para que el programa sea escalable y fácil de mantener. Otra gente los ha usado y les ha ido bien.
- Ayudan a reutilizar técnicas.
- Mucha gente los conoce y ya sabe cómo aplicarlos. Están en un alto nivel de abstracción.
- El diseño se puede aplicar a diferentes situaciones.

Existen dos **modelo básicos de programas concurrentes**:

- Un programa resulta de la actividad de objetos activos que interaccionan entre sí directamente o a través de recursos y servicios pasivos.

- Un programa resulta de la ejecución concurrente de tareas. Cada tarea es una unidad de trabajo abstracta y discreta que idealmente puede realizarse con independencia de las otras tareas.

**No es obligatorio** utilizar patrones, solo es aconsejable en el caso de tener el mismo problema o similar que soluciona el patrón, siempre teniendo en cuenta que en un caso particular puede no ser aplicable.

Como podemos ver en la siguiente web, lo normal es que encontremos la definición de los patrones de diseño en UML (**Unified Modeling Language** – Lenguaje Unificado de Modelado).

## 6.2. Documentación

Para la documentación de nuestras aplicaciones tendremos en cuenta:

Hay que añadir **explicaciones a todo lo que no es evidente**. Pero, no hay que repetir lo que se hace, sino explicar **por qué se hace**.

Documentando nuestro código responderemos a estas preguntas:

- ¿De qué se encarga una clase? ¿Un paquete?
- ¿Qué hace un método? ¿Cuál es el uso esperado de un método?
- ¿Para qué se usa una variable? ¿Cuál es el uso esperado de una variable?
- ¿Qué algoritmo estamos usando? ¿De dónde lo hemos sacado?
- ¿Qué limitaciones tiene el algoritmo? ¿... la implementación?
- ¿Qué se debería mejorar ... si hubiera tiempo?

En la siguiente tabla tenemos un resumen de los distintos tipos de comentarios en Java:

**Tipos de comentarios en Java**

| | Javadoc | Una línea | Tipo C |
|---|---|---|---|
| **Sintáxis** | Comienzan con "/**", se pueden prolongar a lo largo de varias líneas (que probablemente comiencen con el carácter "*") y terminan con los caracteres "*/". Cuenta con etiquetas específicas tipo: **@author**, **@param**, **@return** | Comienzan con "//" y terminan con la línea. | Comienzan con "/*", se pueden prolongar a lo largo de varias líneas (que probablemente comiencen con el carácter "*") y terminan con los caracteres "*/". |
| **Propósito** | Generar documentación externa. | Documentar código que no necesitamos que aparezca en la documentación externa. | Eliminar código. |
| **Uso** | **Obligado:**<br>✓ Al principio de cada clase.<br>✓ Al principio de cada método.<br>✓ Antes de cada variable de clase. | ✓ Al principio de fragmento de código no evidente.<br>✓ A lo largo de los bucles.<br>✓ Siempre que hagamos algo raro o que el código no sea evidente. | Ocurre a menudo que código obsoleto no queremos que desaparezca, sino mantenerlo "por si acaso". Para que no se ejecute, se comenta. |

**Además** de lo anterior, al **documentar** nuestras aplicaciones concurrentes destacaremos:

- Las **condiciones de sincronismo** que se hayan implementado en la clase o método (en la documentación javadoc).

- Destacaremos las **regiones críticas** que hayamos identificado y el **recurso que compartido** a proteger.

# 7. Programación paralela y distribuida.

Dos **procesos se ejecutan de forma paralela**, si las **instrucciones** de ambos se están **ejecutando** realmente de **forma simultánea**. Esto sucede en la actualidad en sistemas que poseen más de un núcleo de procesamiento.

La **programación paralela y distribuida** consideran los **aspectos conceptuales y físicos** de la computación paralela; siempre con el objetivo de **mejorar las prestaciones aprovechado la ejecución simultánea de tareas**.

Tanto en la programación paralela como distribuida, existe **ejecución simultánea de tareas que resuelven un problema común**. La **diferencia** entre ambas es:

- La **programación paralela** se centra en **microprocesadores multinúcleo** (en nuestros PC y servidores); o sobre los llamados **supercomputadores**, fabricados con arquitecturas específicas, compuestos por gran cantidad de equipos **idénticos** interconectados entre sí, y que cuentan son sistemas operativos propios.

- La **programación para distribuida**, en sistemas formados por un conjunto de ordenadores **heterogéneos interconectados** entre sí, por **redes de comunicaciones de propósito general**: redes de área local, metropolitana; incluso, a través de Internet. Su **gestión** se realiza utilizando **componentes, protocolos estándar y sistemas operativos de red**.

En la **computación paralela y distribuida**:

- **Cada procesador** tiene asignada la tarea de resolver una **porción del problema**.
- En programación paralela, los **procesos** pueden **intercambiar datos**, a través de direcciones de **memoria compartidas o** mediante una **red de interconexión propia**.
- En programación distribuida, el **intercambio de datos y la sincronización** se realizará mediante **intercambio de mensajes**.
- El **sistema** se presenta como una unidad **ocultando la realidad de las partes que lo forman**.

## 7.1. Conceptos básicos.

Comencemos revisando algunas clasificaciones de sistemas distribuidos y paralelos:

- En función de los **conjuntos de instrucciones y datos** (conocida como la **taxonomía de Flynn**):
  - La arquitectura secuencial la denominaríamos SISD (single instruction single data).
  - Las diferentes arquitecturas paralelas (o distribuidas) en diferentes grupos: SIMD (single instruction multiple data), MISD (multiple instruction single data) y MIMD (multiple instruction multiple data), con algunas variaciones como la SPMD (single program multiple data).

- Por **comunicación y control**:
  - **Sistemas de multiprocesamiento simétrico** (SMP). Son MIMD con memoria compartida . Los procesadores se comunican a través de esta memoria compartida; este es el caso de los microprocesadores de múltiples núcleos de nuestros PCs.
  - **Sistemas MIMD con memoria distribuida** . La memoria está distribuida entre los procesadores (o nodos) del sistema, cada uno con su propia memoria local, en la que poseen su propio programa y los datos asociados. Una red de interconexión conecta los procesadores (y sus memorias locales), mediante enlaces (links) de comunicación, usados para el intercambio de mensajes entre los procesadores. Los procesadores intercambian datos entre sus memorias cuando se pide el valor de variables remotas. Tipos específicos de estos sistemas son:
    - Clusters. Consisten en una colección de ordenadores (no necesariamente homogéneos) conectados por red para trabajar concurrentemente en tareas del mismo programa. Aunque la interconexión puede no ser dedicada.
    - Grid. Es un cluster cuya interconexión se realiza a través de internet.

Veamos una introducción a algunos **conceptos básicos** que debemos conocer para **desarrollar en sistemas distribuidos**:

- **Distribución**: construcción de una aplicación por partes, a cada parte se le asigna un conjunto de responsabilidades dentro del sistema.
- **Nudo de la red**: uno o varios equipos que se comportan como una unidad de asignación integrada en el sistema distribuido.
- Un **objeto distribuido** es un **módulo de código con plena autonomía** que se puede instanciar en cualquier nudo de la red y a cuyos servicios pueden acceder clientes ubicados en cualquier otro nudo.
- **Componente: Elemento de software que encapsula una serie de funcionalidades.** Un componente, es una unidad independiente, que puede ser utilizado en conjunto con otros componentes para formar un sistema más complejo (concebido por ser **reutilizable**). Tiene especificado: los **servicios** que ofrece; los

**requerimientos** que necesarios para poder ser instalado en un nudo; las **posibilidades de configuración** que ofrece; y, **no está ligado** a ninguna aplicación, que se puede instanciar en cualquier nudo y ser **gestionado por herramientas automáticas**. Sus **características**:

- **Alta cohesión**: todos los elementos de un componente están estrechamente relacionados.
- **Bajo acoplamiento**: nivel de independencia que un componente respecto a otros.

- **Transacciones: Conjunto de actividades que se ejecutan en diferentes nudos de una plataforma distribuida para ejecutar una tarea de negocio.** Una transacción finaliza cuando todas las parte implicadas clientes y múltiples servidores confirman que sus correspondientes actividades han concluido con éxito. **Propiedades ACID** de una transacción:
  - **Atomicidad**: Una transacción es una unidad indivisible de trabajo, Todas las actividades que comprende deben ser ejecutadas con éxito.
  - **Congruencia**: Después de que una transacción ha sido ejecutada, la transacción debe dejar el sistema en estado correcto. Si la transacción no puede realizarse con éxito, debe restaurar el sistema al estado original previo al inicio de la transacción.
  - **Aislamiento**: La transacciones que se ejecutan de forma concurrente no deben tener interferencias entre ellas. La transacción debe sincronizar el acceso a todos los recursos compartidos y garantizar que las actualizaciones concurrentes sean compatibles entre si.
  - **Durabilidad**: Los efectos de una transacción son permanentes una vez que la transacción ha finalizado con éxito.
- **Gestor de transacciones.** Controla y supervisa la ejecución de transacciones, asegurando sus propiedades ACID.

Las **mejoras arquitectónicas** que han sufrido los computadores se han basado en la **obtención de rendimiento explotando los diferentes niveles de paralelismo**.

En un sistema podemos encontrar los siguientes niveles de paralelismo:

- A nivel de **bit**. Conseguido incrementando el tamaño de la palabra del microprocesador. Realizar operaciones sobre mayor número de bits. Esto es, el paso de palabras de 8 bits, a 16, a 32 y en los microprocesadores actuales de 64 bits.
- A nivel de **instrucciones**. Conseguida introduciendo pipeline en la ejecución de instrucciones máquina en el diseño de los microprocesadores.
- A nivel de **bucle**. Consiste en dividir las interaciones de un bucle en partes que se pueden realizar de manera complementaria. Por ejemplo, un bucle de 0 a 100; puede ser equivalente a dos bucles, uno de 0 a 49 y otro de 50 a 100.
- A nivel de **procedimientos**. Identificando qué fragmentos de código dentro de un programa pueden ejecutarse de manera simultánea sin interferir la tarea de una en la otra.

- A nivel de **tareas dentro de un programa**. Tareas que cooperan para la solución del programa general (utilizado en sistemas distribuidos y paralelos).
- A nivel de **aplicación dentro de un ordenador**. Se refiere a los conceptos que vimos al principio de la unidad, propios de la gestión de procesos por parte del sistema operativo multitarea: planificador de procesos, Round-Robin, quamtum, etc.

En sistemas distribuidos hablaremos del **tamaño de grano o granularidad**, que es una **medida de la cantidad de computación** de un proceso software. Y se considera como el **segmento de código escogido para su procesamiento paralelo**.

- Paralelismo de **Grano Fino**: No requiere tener mucho conocimiento del código, la paralelización se obtiene de forma casi automática. Permite obtener buenos resultados en eficiencia en poco tiempo. Por ejemplo: la descomposición de bucles.
- Paralelismo de **Grano Grueso**: Es una paralelización de alto nivel, que engloba al grano fino. Requiere mayor conocimiento del código, puesto que se paraleliza mayor cantidad de él. Consigue mejores rendimientos, que la paralelización fina, ya que intenta evitar los overhead (exceso de recursos asignados y utilizados) que se suelen producir cuando se divide el problema en secciones muy pequeñas. Un ejemplo de paralelismo grueso lo obtenemos descomponiendo el problema en dominios (dividiendo conjunto de datos a procesar, acompañando a estos, las acciones que deban realizarse sobre ellos).

## 7.2. Modelos de infraestructura para programación distribuida.

Las aplicaciones distribuidas requieren que componentes que se ejecutan en diferentes procesadores se comuniquen entre sí. Los **modelos** de infraestructura que permiten la **implementación** de esos **componentes**, son:

- **Uso de Sockets**: Facilitan la generación dinámica de canales de comunicación. Es actualmente la base de la comunicación. Pero al ser de muy bajo nivel de abstracción, no son adecuados a nivel de aplicación.
- **Remote Procedure Call (RPC)**: Abstrae la comunicación a nivel de invocación de procedimientos. Es adecuada para programación estructurada basada en librerías.
- Invocación remota de objetos: Abstrae la comunicación a la invocación de métodos de objetos que se encuentran distribuidos por el sistema distribuido. Los objetos se localizan por su identidad. Es adecuada para aplicaciones basadas en el paradigma OO.
- **RMI (Remote Method Invocation)** es la solución Java para la comunicación de objetos Java distribuidos. Presenta un inconveniente, y es el paso de parámetros por valor implica tiempo para hacer la serialización, enviar los objetos

serializados a través de la red y luego volver a recomponer los objetos en el destino.

- **CORBA** (Common Object Request Brocker Architecture) . Para facilitar el diseño de aplicaciones basadas en el paradigma Cliente/Servidor. Define servidores estandarizados a través de un modelo de referencia, los patrones de interacción entre clientes y servidores y las especificaciones de las APIs.
- **MPI** ("Message Passing Interface", Interfaz de Paso de Mensajes) es un estándar que define la sintaxis y la semántica de las funciones contenidas en una biblioteca de paso de mensajes diseñada para ser usada en programas que exploten la existencia de múltiples procesadores.
- La Interfaz de Paso de Mensajes es un protocolo de comunicación entre computadoras. Es el estándar para la comunicación entre los nodos que ejecutan un programa en un sistema de memoria distribuida. Las llamadas de MPI se dividen en cuatro clases:
  - Llamadas utilizadas para inicializar, administrar y finalizar comunicaciones. Llamadas utilizadas para transferir datos entre un par de procesos.
  - Llamadas para transferir datos entre varios procesos.
  - Llamadas utilizadas para crear tipos de datos definidos por el usuario.
- **Máquina paralela virtual**. Paquetes software que permite ver el conjunto de nodos disponibles como una máquina virtual paralela ; ofreciendo una opción práctica, económica y popular hoy en día para aproximarse al cómputo paralelo .