1. Programación multiproceso y paralela.

El uso de **varios procesadores** dentro de un sistema informático se nombra multiproceso.

1.1 Programación multiproceso.

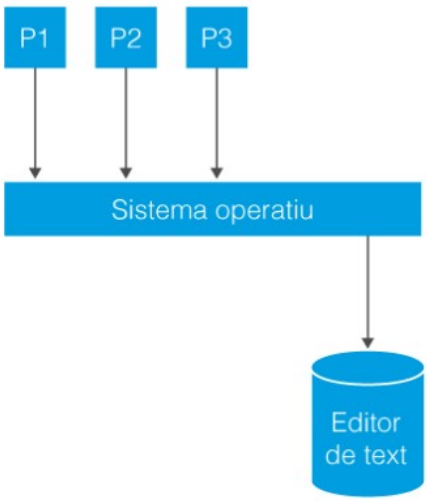
1.1.1 Proceso, sistemas monoprocesador y multiprocesador.

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

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

El **sistema operativo** es el encargado de la gestión de procesos.

FIGURA 1.1. Execució de processos



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

El elemento de hardware en el que se ejecutan los procesos es el **procesador**.

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

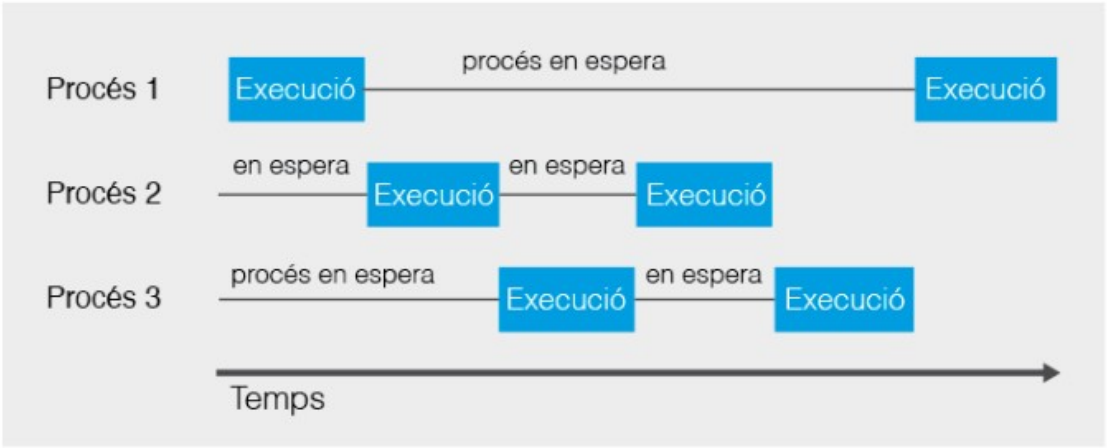
Un sistema **monoprocesador** es aquel que está formado únicamente por un procesador.

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

Actualmente, la mayoría de los sistemas operativos aprovechan los tiempos de reposo de los procesos, cuando esperan, por introducir al procesador otro proceso, **simulando así una ejecución paralela**.

De forma genérica llamaremos a los procesos que se ejecutan a la vez, ya sea de forma real o simulada, **procesos concurrentes**.

FIGURA 1.2. Execució de processos concurrents



Es el sistema operativo el encargado de gestionar la ejecución concurrente de diferentes procesos contra un mismo procesador y a menudo nos referimos con el nombre de **multiprogramación**.

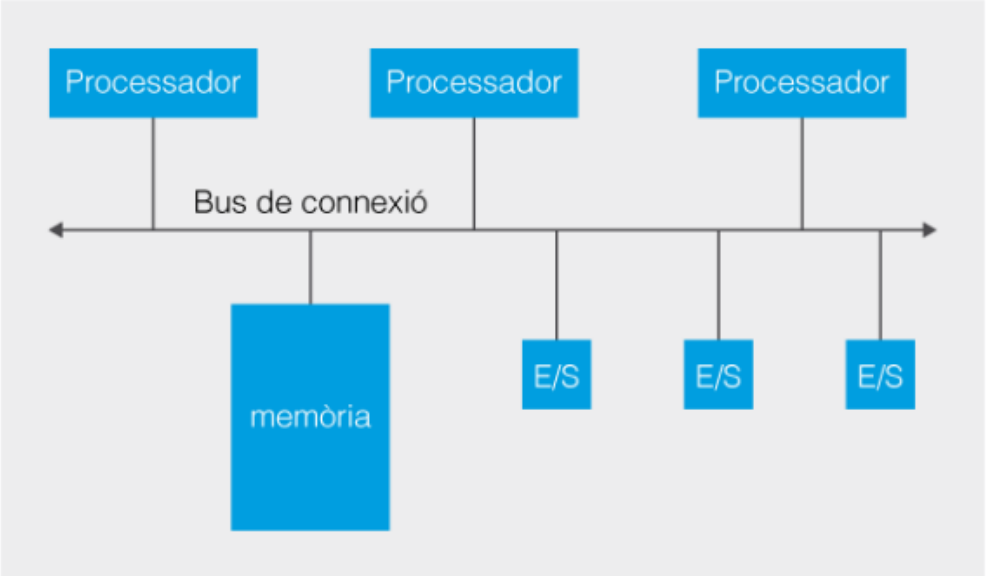
Hablamos de multiprogramación cuando el sistema operativo gestiona la ejecución de procesos concurrentes a un sistema monoprocesador.

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

En un sistema informático multiprocesador existen dos o más procesadores, por tanto se pueden ejecutar simultáneamente varios procesos. También se pueden calificar de multiprocesadores aquellos dispositivos cuyo procesador tiene más de un núcleo (multicore) , es decir, tienen más de una CPU al mismo circuito integrado del procesador.

Sistemas multiprocesadores fuertemente acoplados: en esta arquitectura los diferentes procesadores comparten una misma memoria y están interconectados a ella a través de un bus.

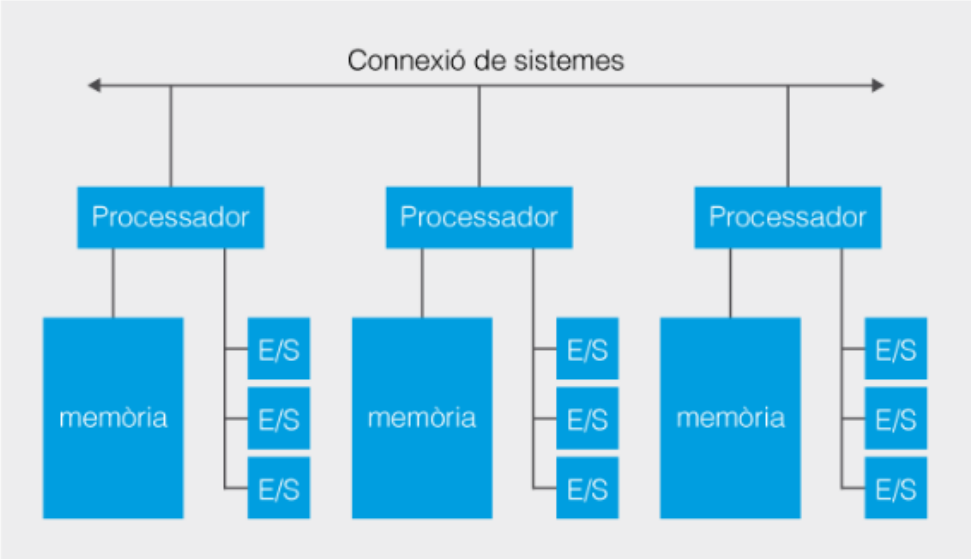
FIGURA 1.3. Sistema informàtic multiprocessador fortament acoblat



Se pueden dividir en **sistemas multiproceso simétricos**, en los que los procesadores del sistema son de características similares y compiten entre iguales por ejecutar procesos, y **sistemas multiproceso asimétricos** en los cuales uno de los procesadores del sistema, el máster, controla la resto de procesadores. Este sistema también se llama master-slave.

Sistemas multiprocesadores débilmente acoplados: estos sistemas no comparten memoria, cada procesador tiene una memoria asociada.

FIGURA 1.4. Sistema informàtic multiprocessador dèbilment acoblat



Un tipo de estos sistemas poco acoplados son los **sistemas distribuidos**, en los que cada dispositivo monoprocesador (o multiprocesador) podría estar situado en lugares físicamente distantes.

1.1.2 Programación concurrente, paralela y distribuida.

La ejecución simultánea de procesos se nombra también concurrencia.

No quiere decir que se hayan de ejecutar exactamente al mismo momento, el intercalado de procesos también se considera ejecución concurrente.

Hablamos de **programación concurrente** cuando se ejecutan en un dispositivo informático de forma simultánea diferentes tareas (procesos).

La ejecución concurrente puede comportar ciertos problemas a la hora de acceder a las datos. Básicamente usaremos dos técnicas por evitarlos: **el boqueo y la comunicación**.

Si la programación concurrente se da en un computador con un único procesador hablaremos de multiprogramación. En cambio, si el computador tiene más de un procesador y, por tanto, los procesos se pueden ejecutar de forma realmente simultánea, hablaremos de programación paralela.

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

Cuando la programación concurrente se realiza en un sistema multiprocesador hablamos de **programación paralela**.

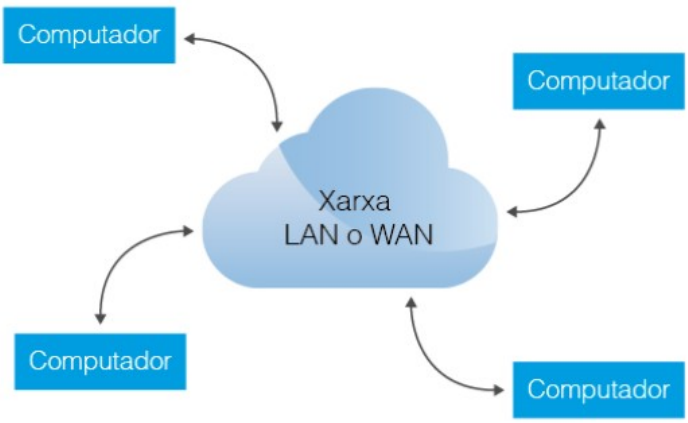
FIGURA 1.5. Execució en paral·lel

El principal desventaja de la programación paralela son los controles que debemos añadir para que la comunicación y sincronización de los procesos que se ejecutan concurrentemente sean correctos.

El lenguaje de programación Java tiene integrado el apoyo a la concurrencia en el propio lenguaje y no a través de librerías.

Un tipo especial de programación paralela es la renombrada programación distribuida. Esta programación se da en sistemas informáticos distribuidos. Un sistema distribuido está formado por un conjunto de ordenadores que pueden estar situados en lugares geográficos diferentes unidos entre ellos a través de una red de comunicaciones.

FIGURA 1.6. Execució de programació distribuïda



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

En los sistemas distribuidos, a diferencia de los sistemas paralelos, no existe memoria compartida, por tanto si necesitan la utilización de variables compartidas se crearán réplicas de estas variables a los diferentes dispositivos de la red y habrá que controlar la coherencia de las datos.

Los principales ventajas son que se trata de sistemas altamente escalables y por tanto reconfigurables fácilmente y de alta disponibilidad. Como desventajas más importantes, encontraremos que son sistemas heterogéneos, complejos de sincronizar. Las comunicaciones se hacen a través de mensajes que utilizan la red de comunicación compartida y eso puede provocar la saturación.

1.1.3 Ventajas y desventajas del multiproceso.

VENTAJAS

Incrementar la potencia de cálculo y el rendimiento es uno de los principales ventajas.

A veces las técnicas de sincronismo o comunicación son más costosas en tiempo que la ejecución de los procesos.

Un sistema multiprocesador es **flexible**, ya que, si aumentan los procesos que se están ejecutando es capaz de distribuir la carga de trabajo de los procesadores, y puede también reasignar dinámicamente los recursos de memoria y los dispositivos por ser más eficientes.

Es de **fácil crecimiento**. Si el sistema lo permite, se pueden añadir nuevos procesadores de forma sencilla y así aumentan su potencia.

En el último caso hablaremos de sistemas con una **alta tolerancia a fallos**. Una fallo de un procesador no hace que el sistema se paré.

Los sistemas multiproceso nos permiten diferenciar procesos por su **especialización** y, por tanto, reservar procesadores para operaciones complejas, aprovechar otros para procesamientos paralelos y para avanzar la ejecución.

INCONVENIENTES

Los inconvenientes del multiprocesamiento vienen provocados sobre todo por el **control** que se debe realizar cuando hay varios procesos en ejecución y deben compartir información o comunicarse. Eso incrementa la **complejidad** de la programación y penaliza el tiempo de ejecución.

Si el multiprocesamiento es en un entorno multiprocesador paralelo o distribuido, el tráfico de los buses de comunicación se incrementa paralelamente al número de procesadores o computadores que incorporamos al sistema, y eso puede llegar a ser un crítico **cuello de botella**.

1.2 Procesos y servicios.

Ej: antivirus.
Este proceso que controla las infecciones de todos los ficheros que entran al nuestro ordenador es un **servicio**.

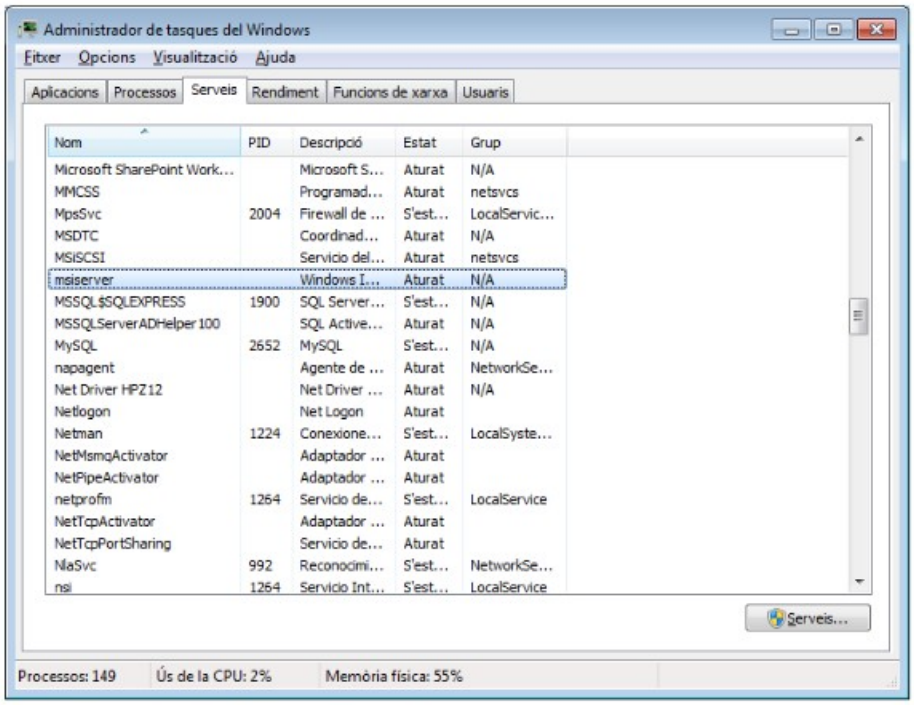
Un **servicio** es un tipo de proceso que no tiene interfaz con el usuario. Pueden ser, dependiendo de su configuración, inicializados por el sistema de forma automática, en el que van realizando sus funciones sin que el usuario se enteré o bien se pueden mantener a la espera de que alguien les haga una petición para realizar una tarea en concreto.

Dependiendo de como se están ejecutando, los procesos se clasificarán como procesos en primero plan (foreground, en inglés) y procesos en segundo plan (background).

Los procesos en **primero plano** mantienen una comunicación con el usuario, ya sea informando de las acciones realizadas o esperando sus peticiones mediante alguna interfaz de usuario. En cambio, los que se ejecutan en **segundo plano** no se muestran explícitamente al usuario, bien porque no es necesaria su intervención o bien porque los datos requeridos no se incorporan a través de una interfaz de usuario.

Los servicios son procesos ejecutados en segundo plano.

FIGURA 1.7. Serveis Windows



Servicio es una nomenclatura utilizada en Windows. En sistemas Linux se llaman también procesos deamon.

El nombre deamon proviene de las siglas inglesas DAEMON (Disk And Execution Monitor). A menudo a la literatura técnica se ha extendido este concepto usando la palabra “demonio”, por el que podéis encontrar esta palabra haciendo referencia en los programas que se ejecutan en segundo plano.

En Linux el nombre de los daemons siempre acaba con la letra d, como por ejemplo httpd, el daemon del http.

Como los servicios no disponen de interfaz de usuario directa, almacenan la información generada o los posibles errores que vayan produciendo, en ficheros de registro habitualmente conocidos como logs.

Algunos servicios pueden estar a la espera de ser llamados por realizar sus funciones. Por ejemplo, un servidor web tiene un servicio activo.

Los servicios pueden estar en ejecución local, en nuestro ordenador, como el Firewall, el antivirus, la conexión Wi-Fi, o los procesos que controlan nuestro ordenador, o bien pueden ejecutarse en algún ordenador de un sistema informático distribuido o en Internet, etc. Las llamadas a los servicios distribuidos se realizan sobre protocolos de comunicación. Por ejemplo, los servicios de HTTP, DNS (Domain Name System), FTP (File Transfer Protocol) son servicios que nos ofrecen servidores que están en Internet.

1.2.1 Hilos y procesos.

Ejemplo: cocinero=procesador; receta=programa; paella=recurso; ingredientes=datos; parte de un plato= tarea / subproceso / hilo.

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

El sistema operativo puede mantener en ejecución varios procesos a la vez haciendo servir concurrencia o paralelismo.

Además, dentro de cada proceso pueden ejecutarse varios hilos, de manera que bloques de instrucciones que presentan cierta independencia puedan ejecutarse a la vez.

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

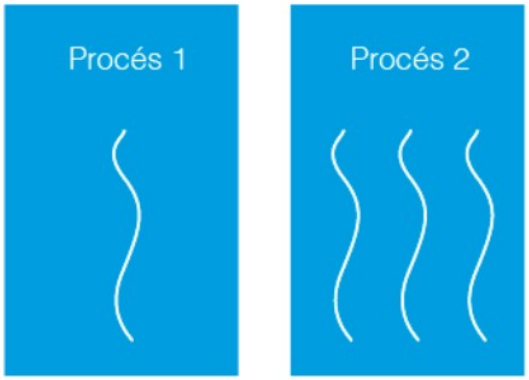
Los procesos son llamados **entidades pesadas** porque están en espacios de direccionamento de memoria independientes, de creación y de comunicación entre procesos, cosa que consume muchos recursos de procesador. En cambio, ni la creación de hilos ni la comunicación consumen mucho tiempo de procesador. Por este motivo los hilos se llaman **entidades ligeras**.

Un proceso estará en ejecución mientras algún de sus hilos estuve activo. Aun así, también es cierto que si finalizamos un proceso de forma forzada, sus hilos también acabarán la ejecución.

Podemos hablar de dos niveles de hilos: los **hilos de nivel de usuario**, que son los que creamos cuando programamos con un lenguaje de programación como Java; y los **hilos de segundo nivel** que son los que crea el sistema operativo por dar apoyo a los primeros.

En referencia al procesamiento, cada hilo se procesa de forma independiente a pesar de que pertenezca o no a un mismo proceso.

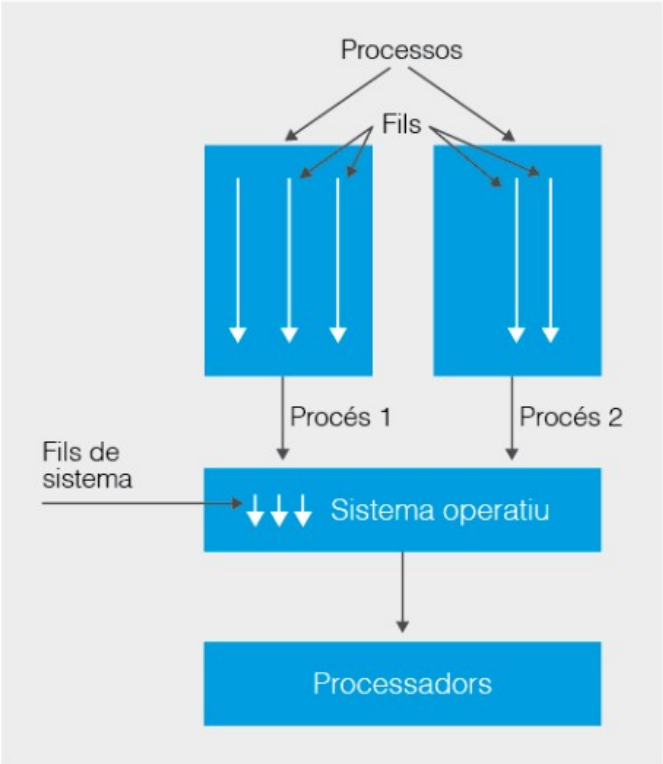
FIGURA 1.8. Procés amb un únic fil en execució i procés amb diversos fils en execució



Para hilos de un mismo proceso el sistema operativo debe mantener los mismos datos en memoria.

En caso de que sea necesario que un mismo procesador alterne la ejecución de varios procesos, habrá que restaurar la memoria específica del proceso en cada cambio. Esto se llama **cambio de contexto**. Si la alternancia de procesamiento se hace entre hilos de un mismo proceso, no hará falta restaurar la memoria y por tanto el proceso será muy más eficiente.

FIGURA 1.9. Dos processos amb diferents fils en execució. Cada fil s'executa de forma independent. Els fils del mateix procés comparteixen memòria.



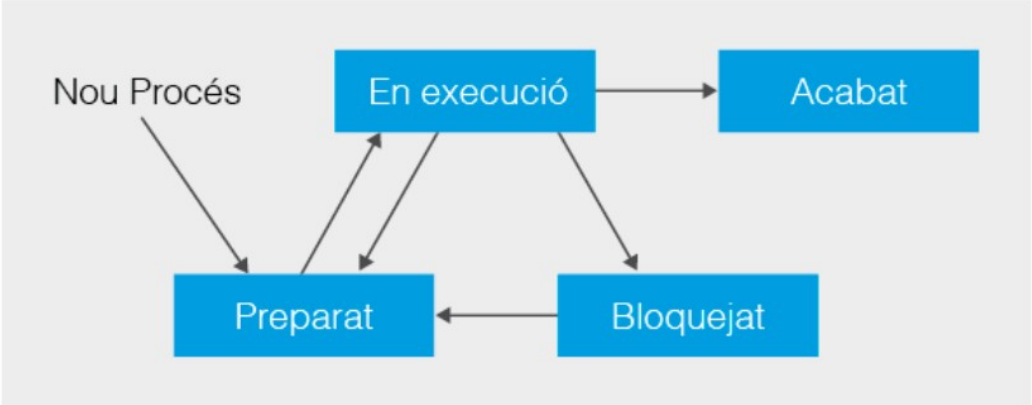
1.3 Gestión de procesos y desarrollo de aplicaciones con finalidad de computación paralela.

El tiempo desde que un proceso inicia la ejecución hasta que la finaliza se llama **ciclo de vida del proceso**. Para que el sistema pueda gestionar las necesidades de los procesos durante todo su ciclo de vida, los identificará etiquetándolos con un estado que indiqué las necesidades de procesamiento.

1.3.1 Estados de un proceso.

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

FIGURA 1.10. Estats d'un procés



El primer estado a que nos encontramos es **nuevo**, es decir, cuando un proceso es creado. Una vez creado, pasa en el estado de **preparado**. En este momento el proceso está preparado para hacer uso del procesador, está compitiendo por el recurso del procesador. El planificador de procesos del sistema operativo es el que decide cuando entra el proceso a ejecutarse. Cuando el proceso se está ejecutando, su estado se llama **en ejecución**. Otra vez, es el planificador el encargado de decidir cuando abandona el procesador.

El planificador de procesos sigue una política que indica cuando un proceso debe cambiar de estado, cuando debe dejar libre o entrar a hacer uso del procesador.

Posible politica del planificador de procesos: asignar diferentes turnos de ejecución. Es decir, asignar un tiempo de ejecución fijo a cada proceso y una vez acabado este tiempo cambiar el proceso del estado de ejecución al estado de **preparado**. Esperando ser asignado de nuevo al procesador.

Cuando un proceso abandone el procesador, porque el planificador de procesos así la habrá decidido, cambiará al estado de **preparado** y se quedará esperando a que el planificador le asigné el turno para conseguir de nuevo el procesador y volver en el estado de **ejecución**.

Desde el estado de ejecución, un proceso puede pasar en el estado de **bloqueado o en espera**. En este estado, el proceso estará a la espera de un acontecimiento, como puede ser una operación de entrada/salida, o la espera de la finalización de otro proceso, etc. Cuando el acontecimiento esperado suceda, el proceso volverá en el estado de **preparado**, guardando el turno con la resto de procesos preparados.

El último estado es **finalizado/acabado**. Es un estado al que se llega una vez el proceso ha finalizado/terminado toda su ejecución y estará a punto para que el sistema liberé cuando pueda los recursos asociados.

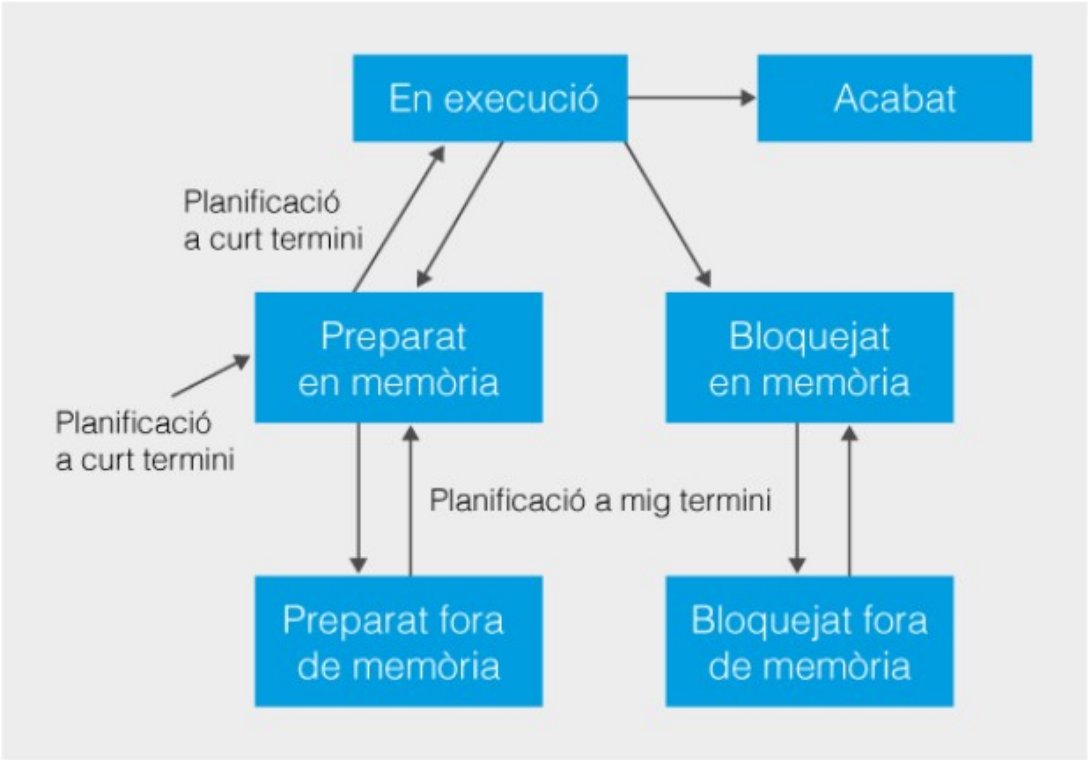
Cada vez que un proceso cambia al estado de ejecución, se llama **cambio de contexto**, porque el sistema operativo debe almacenar los resultados parciales conseguidos durante la ejecución del **proceso saliente** y es necesario que cargué los resultados parciales de la última ejecución del **proceso entrante** en los registros del procesador, asegurando la accesibilidad a las variables y al código que toque seguir ejecutando.

1.3.2 Planificación de procesos.

La planificación de procesos es un conjunto de protocolos o políticas que deciden en qué orden se ejecutarán los procesos al procesador. Estas políticas siguen algunos criterios de prioridad según la importancia del proceso, el tiempo de utilización del procesador, el tiempo de respuesta del proceso, etc.

La función de planificación (scheduler en inglés) del núcleo del sistema operativo, es la encargada de decidir qué proceso entra al procesador, puede dividirse en tres niveles:

FIGURA 1.11. Planificació de processos



1. **Planificación a largo plazo**: en el momento que se crea un proceso se deciden algunos criterios de planificación, como la unidad de tiempo máximo que un proceso podrá permanecer en ejecución (que se nombra **quantum**) o la prioridad inicial que se le asignará a cada proceso de forma dinámica por la utilización del procesador.

2. **Planificación a corto plazo**: cada vez que un proceso abandona el procesador hay que tomar la decisión de cuál será el nuevo proceso que entrará a hacer uso. En este caso las políticas intentan minimizar el tiempo necesario para conseguir un cambio de contexto. Los procesos recientemente ejecutados suelen tener prioridad pero hay que asegurar que nunca se excluí permanentemente ningún proceso de la ejecución.

3. **Planificación a medio plazo**: existen otros partes del sistema operativo que también son importantes a la planificación de procesos. En el caso del **SWAP de memoria**, si se quita un proceso de la memoria por problemas de espacio hace que no pueda ser planificable de forma inmediata. El planificador a medio plazo será el encargado de decidir cuando es preciso llevar a la memoria principal las datos de los procesos almacenados fuera y cuáles habrá que trasladar de la memoria principal al espacio de intercambio.

Los sistemas multiprocesadores presentan además dos componentes de planificación específicos.

El componente de **planificación temporal** en el que se encuentra definida la política de planificación que actúa sobre cada procesador de forma individual, como si fuera un sistema monoprocessador. El planificador temporal decide cuando de tiempo ha que dedicar a procesar cada proceso, cuando es preciso hacer cambiar, por qué motivo, etc.

El componente de **planificación espacial**, en el que se define como se reparten los procesos entre los procesadores. Es decir, organiza qué procesador ejecuta qué proceso.

Una política de **afinidad al procesador** puede provocar una sobrecarga en algunos procesadores y hacer que la ejecución no sea demasiado eficiente. Por contra, una política de **reparto de carga** evita esta supuesta infrautilización de algunos procesadores, pero aumenta las sobrecargas a la memoria compartida.

1.3.3 Programación paralela transparente.

Cuando buscamos resultados rápidamente o mucha potencia de cálculo podemos pensar al aumentar el número de procesadores de nuestra computadora y ejecutar una aplicación de forma paralela a los diferentes procesadores. Pero la ejecución en paralelo no es siempre posible. En primer lugar porque hay aplicaciones que no se pueden paralelizar, sino que su ejecución debe ser secuencial. Y en segundo lugar, por la dificultad para desarrollar entornos de programación en sistemas paralelos, ya que se requiere un esfuerzo importante de los programadores.

Java proporciona apoyo a la programación concurrente con librerías de bajo nivel a través de la clase **java.lang.Thread** y de la interfaz **java.lang.Runnable** . Hay problemas que pueden programarse de forma paralela siguiendo patrones semejantes. El uso de Thread y Runnable comporta un esfuerzo adicional del programador, obligándolo a añadir pruebas extras para verificar el comportamiento correcto del código, evitando lecturas y escrituras erróneas, sincronizando las ejecuciones de los procesos y evitando los problemas derivados de los bloqueos.

Una aproximación relativamente sencilla a algunos de estos problemas que pueden resolverse de forma paralela con implantaciones típicas que presentan un cierto nivel de patronazgo son los marcos predefinidos de programación paralela del lenguaje a Java: **Ejecutor** y **fork-join**. Son librerías que encapsulan buena parte de la funcionalidad y con ella de la complejidad adicional que la programación de más bajo nivel (Threads) presenta.

1.4 Sincronización y comunicación entre procesos.

Las técnicas que nos ayudan a controlar la orden de ejecución de los diferentes procesos se denominan técnicas de comunicación y sincronización.

1.4.1 Competencia de recursos y sincronización.

Los procesos concurrentes se pueden clasificar en independientes o cooperantes según su grado de colaboración. Un proceso **independiente** no necesita ayuda ni cooperación de otros procesos. En cambio, los procesos **cooperantes** están diseñados por trabajar conjuntamente con otros procesos y, por tanto, se han de poder comunicar e interactuar entre ellos.

En la programación concurrente el proceso de sincronización permite que los procesos que se ejecutan de forma simultánea se coordinen, parando la ejecución de aquellos que vayan más avanzados hasta que se cumplan las condiciones óptimas para estar seguros que los resultados finales serán correctos.

Por sincronizar procesos podemos hacer servir diferentes métodos:

- **Sincronismo condicional**: o condición de sincronización. Un proceso o hilo se encuentra en estado de ejecución y pasa en estado de bloqueo esperando que una cierta condición se cumpla para continuar su ejecución. Otro proceso hace que esta condición se cumpla y así el primer proceso pasa de nuevo al estado de ejecución.

- **Exclusión mutua**: se produce cuando dos o más procesos o hilos quieren acceder a un recurso compartido. Se deben establecer protocolos de ejecución para que no accedan de forma concurrente al recurso.

- **Comunicación por mensajes**: en la que los procesos se intercambian mensajes para comunicarse y sincronizarse. Es la comunicación típica en sistemas distribuidos y es utilizada también en sistemas no distribuidos.

1.4.2 Sincronización y comunicación.

Secciones críticas

Las secciones críticas son uno de los problemas que con más frecuencia se dan en programación concurrente. Tenemos varios procesos que se ejecutan de forma concurrente y cada uno de ellos tiene una parte de código que se debe ejecutar de forma exclusiva ya que accede a recursos compartidos como ficheros, variables comunes, registros de bases de datos, etc.

La solución pasará por obligar a acceder a los recursos a través de la ejecución de un código que llamaremos sección crítica y que nos permitirá proteger aquellos recursos con mecanismos que impidan la ejecución simultánea de dos o más procesos dentro los límites de la sección crítica.

El algoritmo de sincronización que evita el acceso a una región crítica por más de un hilo o proceso y que garantiza que únicamente un proceso estará haciendo uso de este recurso y el resto de hilos/procesos que quieren utilizarlo estarán a la espera de que sea liberado, se llama **algoritmo de exclusión mutua**.

Exclusión mutua (MUTEX, mutual exclusión en inglés) es el tipo de sincronización que impide que dos procesos ejecuten simultáneamente una misma sección crítica.

Para validar cualquier mecanismo de sincronización de una sección crítica se han de cumplir los siguientes criterios:

- **Exclusión mutua**: no puede haber más de un proceso simultáneamente en la sección crítica.
- **No inanición**: un proceso no puede esperar un tiempo indefinido para entrar a ejecutar la sección crítica.
- **No interbloqueo**: ningún proceso de fuera de la sección crítica puede impedir que otro proceso entre en la sección crítica.
- **Independencia del hardware**: inicialmente no se deben hacer suposiciones con respecto al número de procesadores o la velocidad de los procesos.

Productor-consumidor

El problema productor-consumidor es un ejemplo clásico donde hay que dar un tratamiento independiente a un conjunto de datos que se van generando de forma más o menos aleatoria o cuando menos de una forma en que no es posible predecir en qué momento se generará una dato. Para evitar un uso excesivo de los recursos del ordenador esperando la llegada de datos, los sistema prevé dos tipos de procesos: los productores, encargados de obtener los datos a tratar y los consumidores, especializados al hacer el tratamiento de los datos obtenidos por los productores.

En esta situación el productor genera una serie de datos que el consumidor recoge. Imaginamos que es el valor de una variable que el productor modifica y el consumidor toma para utilizarla. El problema viene cuando el productor produce datos a un ritmo diferente del que el consumidor utiliza/recoge.

El productor crea una dato y cambia la variable a su valor. Si el consumidor va más lento, el productor tiene tiempo a generar un nuevo dato y vuelve a cambiar la variable. Por tanto, el proceso del consumidor ha perdido el valor del primer dato.

En caso de que sea el consumidor el que va más rápido, puede pasar que tome dos veces un mismo dato, ya que el productor no ha tenido tiempo de sustituirlo, o que no encuentre nada para consumir. La situación puede complicarse aún más si disponemos de diversos procesos productores y consumidores.

FIGURA 1.13. Productor-consumidor

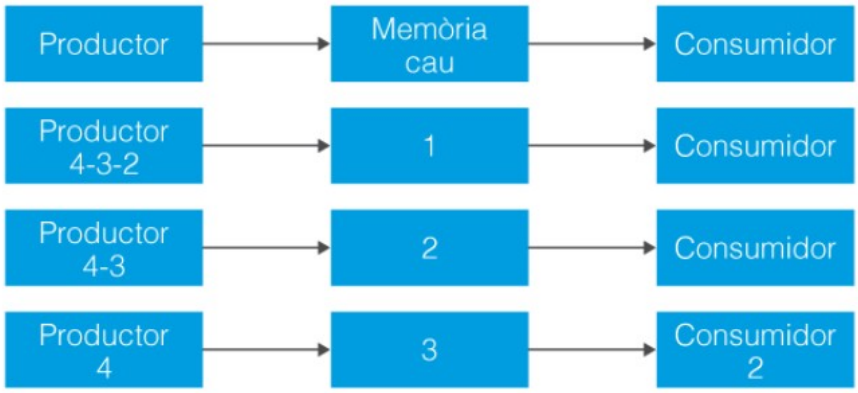


FIGURA 1.14. Buffer

Un buffer es un espacio de memoria para almacenar datos. Se pueden implementar en forma de colas.

Por tanto, debe existir un mecanismo que paré el acceso al dato de los productores y de los consumidores en caso necesario. Una sección crítica. Desgraciadamente no hay suficiente con restringir el acceso a las datos, porque podría darse el caso de que un proceso consumidor esperara la llegada de un dato e impidiera el acceso de los procesos productores de forma que el dato no llegase nunca. Una situación como la descrita se conoce con el nombre de problemas de interbloqueo (deadlock en inglés).

Llamamos **interbloqueo** a la situación extrema a que nos encontramos cuando dos o más procesos están esperando la ejecución del otro para poder continuar de manera que nunca conseguirán desbloquearse.

También llamamos **inanición** a la situación que se produce cuando un proceso no puede continuar su ejecución por falta de recursos. Por ejemplo, si hiciéramos crecer el buffer ilimitadamente.

Por solucionar el problema hay que sincronizar el acceso al buffer. Se debe acceder en exclusión mutua para que los productores no alimenten el buffer si este ya está lleno y los consumidores no puedan acceder si está vacío. Pero además habrá que independizar las secciones críticas de acceso de los productores de las secciones críticas de acceso de los consumidores evitando así que se pueda producir el interbloqueo.

Eso obligará a crear un mecanismo de comunicación entre secciones críticas de manera que cada vez que un productor dejé un dato disponible, avisé a los consumidores que puedan quedar esperando y que por lo menos uno de ellos pueda iniciar el procesamiento de la dato.

Lectores-escritores

Otro tipo de problema que aparece en la programación concurrente, es el producido cuando tenemos un recurso compartido entre varios procesos concurrentes como pueden ser un fichero, una base de datos, etc. que va actualizándose periódicamente. En este caso, los procesos lectores no consumen la dato sino que solo lo utilizan y por tanto, se permite su consulta de forma simultánea, a pesar de que no su modificación.

Así, los procesos que accedan al recurso compartido para leer su contenido se llamarán **lectores**. En cambio, los que accedan para modificarlo recibirán el nombre de **escritores**.

Si la tarea de lectores y escritores no se realiza de forma coordinada podría pasar que un lector leyese varias veces el mismo dato o que el escritor modificara el contenido antes de que hubieran leído todos los lectores, o que el dato actualizado por un escritor malbaratase la actualización de otro, etc. Además, la falta de coordinación obliga a los lectores a comprobar periódicamente si los escritores han hecho modificaciones. Esto hará aumentar el uso del procesador y por tanto podría menguar la eficiencia.

Este problema de sincronización de procesos se llama **problema de lectores-escritores**. Para evitarlo hay que asegurar que los procesos escritores acceden de forma exclusiva al recurso compartido y que a cada modificación se avisa a los procesos lectores interesados en el cambio.

Así, los procesos lectores pueden quedar en espera hasta que son avisados de que hay nuevos datos y pueden empezar a leer; de esta manera se evita que los lectores estén constantemente accediendo al recurso sin que el escritor haya puesto ningún dato nuevo, optimizando, de esta manera, los recursos.

1.4.3 Soluciones a problemas de sincronización y comunicación.

A lo largo de la historia se han propuesto soluciones específicas para los problemas anteriores que vale la pena tener en cuenta, a pesar de que resulta difícil generalizar una solución ya que dependen de la complejidad, la cantidad de secciones críticas, del número de procesos que requieran exclusión mutua y de la interdependencia existente entre procesos. Aquí veremos la sincronización mediante semáforos, de monitores y de paso de mensajes.

Semáforos

Los semáforos son una técnica de sincronización de memoria compartida que impide la entrada del proceso a la sección crítica bloqueándolo. El concepto fue introducido por el informático holandés **Dijkstra** para resolver el problema la exclusión mutua y permitir resolver gran parte de los problemas de sincronización entre procesos.

Los semáforos, no solo controlan el acceso a la sección crítica sino que además disponen de información complementaria por poder decidir si es preciso o no bloquear el acceso de aquellos procesos que lo soliciten. Así por ejemplo, serviría por solucionar problemas sencillos (con poca interdependencia) del tipo escritores-lectores o productores-consumidores.

La solución por ejemplo en el caso de los escritores-lectores pasaría por hacer que los **lectores** antes de consultar la sección crítica pidieran permiso de acceso al semáforo, el cual en función de sí se encuentra bloqueado (rojo) o liberado (verde) parará la ejecución del proceso solicitante o bien le dejará continuar.

Los **escritores** por otra parte, antes de entrar a la sección critica, manipularán el semáforo poniéndolo en rojo y no lo volverán a poner en verde hasta que hayan acabado de escribir y abandonen la sección crítica.

De forma genérica distinguiremos 3 tipos de operaciones en un semáforo:

El semáforo admite 3 operaciones:

- **Inicializa** (initial): se trata de la operación que permite poner en marcha el semáforo. La operación puede recibir un valor por parámetro que indicará si este empezará boqueado (rojo) o liberado (verde).

- **Libera** (sendSignal): cambia el valor interno del semáforo poniéndolo en verde (libera). Si existen procesos en espera, los activa para que finalicen su ejecución.

- **Bloquea** (sendWait): sirve para indicar que el proceso actual quiere ejecutar a sección crítica. En caso de que el semáforo se encuentre bloqueado, se para la ejecución del proceso. También permite indicar que hay que poner el semáforo en rojo.

En caso de que se necesité exclusión mutua, el semáforo dispondrá también de un sistema de espera (mediante colas de procesos) que garantice el acceso a la sección crítica de un único proceso a la vez.

Además de resolver problemas de tipo productor-consumidor o lector-escritor, podemos hacer servir semáforos por gestionar problemas de sincronización donde un proceso tenga que activar la ejecución de otro o de exclusión mutua asegurando que solo un proceso conseguirá acceder a la sección crítica porque el semáforo quedará bloqueado hasta la salida.

Otro ejemplo que puede ilustrar el uso de semáforos, sería el de una oficina bancaria que gestiona nuestro cuenta corriente al que se puede acceder desde diferentes oficinas para ingresar dinero o sacar dinero. Estas dos operaciones modifican nuestro saldo. Serían dos funciones como las siguientes:

```
public void ingresar(float dinero) {
     float aux;
     aux=llegirSaldo();
     aux=aux+dinero;
     saldo=aux;
     guardarSaldo(saldo);
     }


public void sacar(float dinero) {
     float aux;
     aux=llegirSaldo();
     aux=aux – dinero;
     saldo=aux;
     guardarSaldo(saldo);
     }
```

Para evitar el problema podemos utilizar un semáforo. Le iniciaremos a 1, indicando el número de procesos que podrán entrar a la sección crítica. Y tanto en el proceso de sacar dinero como en el de ingresar añadiremos un **sendWait()** al inicio de las secciones críticas y un **sendSignal()** al final.

```
public void ingresar(float dinero) {
     sendWait();
     float aux;
     aux=llegirSaldo();
     aux=aux+dinero;
     saldo=aux;
     guardarSaldo(saldo);
     sendSignal();
     }
```

```
     public void sacar(float dinero) {
          sendWait();
          float aux;
          aux=llegirSaldo();
          aux=aux – dinero;
          saldo=aux;
          guardarSaldo(saldo);
          sendSignal();
          }
```

De esta forma cuando un proceso entra a la sección crítica de un método, coge el semáforo. Si es 1, podrá hacer el **sendWait**, por tanto el semáforo se pondrá a 0, cerrado. Y ningún otro proceso no podrá entrar a ninguno de los dos métodos. Si un proceso intenta entrar, encontrará el semáforo a 0 y quedará bloqueado hasta que el proceso que tiene el semáforo envíe un **sendSignal** , ponga el semáforo a 1 y libere el semáforo.

Utilizar semáforos es una forma eficiente de sincronizar procesos concurrentes. Resuelve de forma simple la exclusión mutua. Pero desde el punto de vista de la programación, los algoritmos son complicados de diseñar y de entender, ya que las operaciones de sincronización pueden estar dispersas por el código. Por tanto se pueden cometer errores con facilidad.

Monitors

(pág. 42)

Otra forma de resolver la sincronización de procesos es el uso de monitores. Los monitores son un conjunto de procedimientos encapsulados que nos proporcionan el acceso a recursos compartidos a través de varios procesos en exclusión mutua.

Las operaciones del monitor están encapsuladas dentro de un módulo para protegerlas del programador. Únicamente un proceso puede estar en ejecución dentro de este módulo.

Un monitor se puede ver como una habitación, cerrada con una puerta, que tiene adentro los recursos. Los procesos que quieran utilizar estos recursos tienen que entrar en la habitación, pero con las condiciones que marca el monitor y únicamente un proceso a la vez. El resto que quiera hacer uso de los recursos tendrá que esperar a que salga el que es dentro.

Por su encapsulación, la única acción que tiene que tomar el programador del proceso que quiera acceder al recurso protegido es informar al monitor. La exclusión mutua está implícita. Los semáforos, en cambio, se tienen que implementar con una secuencia correcta de señal y esperar para no bloquear el sistema.

Un monitor es un algoritmo que hace una abstracción de datos que nos permite representar de forma abstracta un recurso compartido mediante un conjunto de variables que definen su estado. El acceso a estas variables únicamente es posible desde unos métodos del monitor.

El monitor tiene que poder incorporar un mecanismo de sincronización. Por lo tanto, se tienen que implementar. Se puede hacer uso de señales. Estas señales se utilizan para impedir los bloqueos. Si el proceso que está en el monitor tiene que esperar una señal, se pone en estado de espera o bloqueado fuera del monitor, permitiendo que otro proceso haga uso del monitor. Los procesos que están fuera del monitor, están a la espera de una condición o señal para volver a entrar.

Estas variables que se utilizan por las señales y son utilizadas por el monitor para la sincronización se denominan **variables de condición**. Estas pueden ser manipuladas con operaciones de **sendSignal** y **sendWait** (como los semáforos).

- **sendWait**: un proceso que está esperando a un acontecimiento indicado por una variable de condición abandona de forma temporal el monitor y se pone a la cola que corresponde a su variable de condición.

- **sendSignal**: desbloquea un proceso de la cola de procesos bloqueados con la variable de condición indicada y se pone en estado preparado para entrar al monitor. El proceso que entra no tiene que ser el que más tiempo lleva esperando, pero se tiene que garantizar que el tiempo de espera de un proceso sea limitado. Si no existe ningún proceso en la cola, la operación sendSignal no tiene efecto, y el primer proceso que pida el uso del monitor entrará.

Un monitor consta de 4 elementos:

- **Variables o métodos permanentes o privados**: son las variables y métodos internos al monitor que solo son accesibles desde dentro del monitor. No se modifican entre dos llamamientos consecutivos al monitor.

- **Código de inicialización**: inicializa las variables permanentes, se ejecuta cuando el monitor es creado.

- **Métodos externos o exportados**: son métodos que son accesibles desde fuera del monitor por los procesos que quieren entrar a hacer uso.

- **Cola de procesos**: es la cola de procesos bloqueados a la espera de la señal que los libere para volver a entrar al monitor.

En el campo de la programación un monitor es un objeto en el cual todos sus métodos están implementados bajo exclusión mutua. En el lenguaje Java son objetos de una clase en los cuales todos sus métodos públicos son **synchronized**.

Un semáforo es un objeto que permite sincronizar el acceso a un recurso compartido y un monitor es una interfaz de acceso al recurso compartido. Es el encapsulamiento de un objeto, aquello que hace un objeto más seguro, robusto y *escalable.

wait, **notify** y **notifyAll**, son operaciones que permiten sincronizar el monitor.

Sincronizar en Java

En Java la ejecución de objetos no está protegida. Si queremos aplicar exclusión mutua en Java tenemos que utilizar la palabra reservada **synchronized** en la declaración de atributos,de métodos o en un bloque de código dentro de un método.

Todos los métodos con el modificador **synchronized** se ejecutarán en exclusión mutua. El modificador asegura que si se está ejecutando un método sincronizado ninguno otro método sincronizado se pueda ejecutar. Pero los métodos no sincronizados pueden estar ejecutándose y con más de un proceso a la vez. Además, únicamente el método sincronizado puede estar ejecutado por un proceso, el resto de procesos que quieren ejecutar el método se tendrán que esperar a que acabe el proceso que lo está ejecutando.

Método:

```
public synchronized void metodeSincronitzat {

     //parte de código sincronizado,
     //se ejecuta en exclusión mutua

     }
```

Bloque de código:

```
public void metodeNoSincronitzat {
     //part de codi sense sincronitzar

     synchronized(this){
          //part de codi sincronitzat
          }

     //part de codi sense sincronitzar

     }
```

this es el objeto que ha llamado al método, pero también podemos utilizar otro objeto synchroniced(objeto) , si lo que queremos es realizar más de una operación atómica sobre un objeto.

Java, por lo tanto, nos proporciona con **synchronized** los recursos que nos daría la implementación de un **semáforo** o de un **monitor**. Sus librerías nos proporcionan un acceso en exclusión mutua y controlan los errores de bloqueo o interbloqueo que se puedan ocasionar entre los hilos en ejecución.

Atomicidad de una operación y clases atómicas de Java

El único árbitro que tenemos en la gestión de procesos es el planificador. Se encarga de decidir qué proceso hace uso del procesador. Después las instrucciones u operaciones que forman el proceso se van ejecutando en ell procesador siguiendo un orden concreto. Una operación o un grupo de operaciones se tienen que poder ejecutar como si fueran una única instrucción. Además, no se pueden ejecutar de forma concurrente con cualquier otra operación en la cual haya involucrada un dato o recurso que utiliza la primera operación.

El atomicidad de una operación es poder garantizar que no se pueden ejecutar dos operaciones de forma concurrente si hacen uso de un recurso compartido hasta que una de las dos deja libre este recurso. **Una operación atómica únicamente tiene que observar dos estados: el inicial y el resultado**. **Una operación atómica, o se ejecuta completamente o no lo hace en absoluto**.

La **atomicidad real** son las instrucciones que ejecutan en código máquina. En cambio, la **atomicidad modelo** es un grupo de instrucciones, en código máquina, que se ejecuta de forma atómica.

Clases de operaciones atómicas

La instrucción x++; o x= x + 1; tiene la siguiente secuencia de instrucciones atómicas reales:

1. Leer valor x
2. Añadir a x1
3. Guardar valor x

La atomicidad de modelo entiende la instrucción completa, x = x + 1 como atómica.

Dos procesos ejecutan en paralelo el siguiente ejemplo:

```
package operacionsatomiques;

public class OperacionsAtomiques {

     public static void main(String[] args) {
          int x=0;
          x=x+1;
          System.out.println(x);
          }
     }
```

Clases atómicas en Java

Las clases atómicas de Java son una primera solución a los problemas de procesos concurrentes. Permiten ejecutar algunas operaciones como si fueran atómicas reales.

Una operación atómica tiene únicamente dos estados: el inicial y el resultado, es decir, se completa sin interrupciones desde el inicio hasta el final.

En Java, el acceso a las variables de tipos primitivos (excepto **double** y **long**) es atómico. Como también es atómico el acceso a todas las variables de tipo primitivo en las cuales aplicamos el modificador **volatile**.

NOTA: **volatile** es un modificador, utilizado en la programación concurrente, que se puede aplicar a algunos tipos primitivos de variables. Indica que el valor de la variable quedará guardado en la memoria y no en un registro del procesador. Así, es accesible por otro proceso o hilo para modificarla. Por ejemplo: **volatile int c**;

Java asegura la *atomicidad del acceso a las variables volátiles o primitivas, pero no a la suyas operaciones. Esto quiere decir que la operación **x++**; en la cual **x** es un **int** , no es una operación atómica y por tanto no es un hilo seguro.

Java incorpora, desde la versión 1.5, en el paquete **java.util.concurrent.atomic**, las clases que nos permiten convertir en atómicas (hilos seguros) algunas operaciones, como por ejemplo aumentar, reducir, actualizar y añadir un valor. Todas las clases tienen métodos **get** y **set** que también son atómicos.

Las clases que incorpora son: **AtomicBoolean** , **AtomicInteger** , **AtomicIntegerArray** , **AtomicLong** , **AtomicLongArray** y actualizadores que son básicamente derivados de una variable tipo **volatile** que son **AtomicIntegerFieldUpdater** , **AtomicLongFieldUpdater** y **AtomicReferenceFieldUpdater**.

Por lo tanto, podemos convertir la operación x++ o x=x+1 en una operación atómica .

```
public class Comptador {

     private int x = 0;

     public void augmenta() {
          x++;
          }

     public void disminueix() {
          x -- ;
          }

     public int getValorComptador() {
          return x;
          }

     }
```

Para convertir en operaciones atómicas el ejemplo anterior, utilizaremos los métodos de la clase **AtomicInteger** del paquete **java.util.concurrent.atomic.AtomicInteger**.

```
import java.util.concurrent.atomic.AtomicInteger;

public class ContadorAtomic {

     private AtomicInteger x =new AtomicInteger(0);

     public void augmenta() {
          x.incrementAndGet();
          }

     public void disminueix() {
          x.decrementAndGet();
          }

     public int getValorContadorAtomic() {
          return x.get();
          }
          }
```

https://docs.oracle.com/javase/6/docs/api/java/util/concurrent/atomic/package-summary.html

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/atomic/AtomicInteger.html

Colecciones concurrentes

Las colecciones son objetos que contienen múltiples datos de un mismo tipo. Lascolecciones pueden estar involucradas también en la ejecución de procesos concurrentes y ser utilizadas por diferentes procesos de forma que se acaben compartiendo también los datos contenidos a la colección.

En Java para trabajar con colecciones tenemos que utilizar el **Framework Collections** en el cual se encuentran los paquetes **java.util** y **java.util.concurrent**, que contienen las clases e interfaces que nos permiten crear colecciones.

En Java las interfaces **List<E>**, **Set<E>** y **Queue<E>** serian las colecciones más importantes. Representan listas, conjuntos y colas. Map<E> es también una colección de key/valor. Cada una con sus características de ordenación, los métodos de inserción, de extracción i consulta, si permiten elementos repetidos o no, etc.

El problema con estas colecciones es que en la programación concurrente en la cual sean compartidas por diferentes procesos o hilos de forma simultánea, podrían dar resultados inesperados.

Algunos lenguajes de programación solucionaron este problema sincronizando las clases. En Java se utiliza, por ejemplo, **Collections.synchronizedList(List<T> list)** sobre la interfaz **List<E>** para poder sincronizar la clase que la implementa, la clase **ArrayList**. Otras clases que implementan la interfaz **List** es **Vector**. Java ya sincroniza esta clase.

En la versión 1.5 apareció **java.util.concurrent** que incorpora clases que nos permiten solucionar problemas de concurrencia de una forma simple.

ArrayList es una colección de datos y si queremos que sea utilizada por diferentes hilos de ejecución lo que tenemos que hacer es sincronizar todas las operaciones de lectura y escritura para preservar la integridad de los datos. Si este **ArrayList** es muy consultado y poco actualizado, la sincronización penaliza mucho el acceso. Java crea la clase **CopyOnWriteArrayList** , que es la ejecución de hilo seguro de una **ArrayList** .

NOTA:
La clase **CopyOnWriteArrayList** cada vez que se modifica el ***array*** se crea en la memoria un ***array*** con el contenido sin procesos de sincronización. Esto hace que las consultas no sean tan costosas.

La interfaz **BlockingQueue** implementa una cola FIFO, que bloquea el hilo de ejecución si intenta sacar de la cola un elemento y la cola está vacía o si intenta insertar un elemento y está llena. También contiene métodos que hacen que el bloqueo dure un tiempo determinado antes de dar un error de inserción o lectura y así evitar una ejecución infinita.

NOTA:
La cola FIFO (en inglés, First In Firts Out) es una estructura de datos implementada de forma que los primeros elementos al entrar a la estructura son los primeros al salir.

La clase **SynchronousQueue** es una cola de bloqueo, implementa en **BlockingQueue**. Lo que hace es que por cada operación de inserción tiene que esperar una de eliminación y por cada inserción tiene que esperar una eliminación.

ConcurrentMap es otra clase que define una table ***hash*** con operaciones atómicas.

Podéis encontrar más información sobre el paquete en la página oficial de Java.
http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/package-summary.html

Comunicación con mensajes

Una solución a los problemas de concurrencia de procesos, muy adecuada sobre todo para los sistemas distribuidos, son los mensajes. Recordamos que un sistema distribuido no comparte memoria, por lo tanto no es posible la comunicación entre procesos con variables compartidas.

Este sistema de sincronización y comunicación está incluido en la mayoría de sistemas operativos y se usa tanto por la comunicación entre diferentes ordenadores, como para comunicar entre procesos de una mismo ordenador. La comunicación a través de mensajes siempre necesita un proceso emisor y otro de receptor para poder intercambiar información. Por lo tanto, las operaciones básicas serán enviar(mensaje) y recibir(mensaje).

Dado que en esta comunicación pueden estar implicados varios procesos, hay que permitir a las operaciones básicas identificar quién es el destinatario e incluso convendría en ocasiones identificar el emisor. Habrá que extender las operaciones básicas incorporando a quién va dirigido el mensaje o cómo el receptor podría indicar de quién espera el mensaje.

Según la forma de referenciar el destinatario y el emisor, hablaremos de envío de mensajes en **comunicación directa** o **comunicación indirecta**.

En la **comunicación directa**, el emisor identifica el receptor explícitamente cuando envía un mensaje y viceversa, el receptor identifica explícitamente el emisor que le envía el mensaje. Se crea un enlace de comunicación entre el proceso emisor y proceso receptor. Las órdenes de envío y recepción serían:

enviar(A,mensaje) -> Enviar un mensaje al proceso A
recibir(B, mensaje) -> Recibir un mensaje del proceso B

Este sistema de comunicación ofrece una gran seguridad, puesto que identifica los procesos que actúan en la comunicación y no introduce ningún retraso a la hora de identificar los procesos. Por otro lado, esto también es una desventaja, puesto que cualquier cambio en la identificación de procesos implica un cambio en la codificación de las órdenes de comunicación.

La comunicación directa solo es posible de implementar si podemos identificar los procesos. Es decir, hará falta que el sistema operativo y el lenguaje de programación soporten la gestión de procesos. También hace falta que los procesos se ejecuten en un mismo ordenador, puesto que sino es imposible acceder de forma directa.

Cuando los procesos se ejecuten en diferentes ordenadores, será necesario interponer entre ellos un sistema de mensajería que envíe y recoja los mensajes con independencia de que sea el proceso emisor y receptor. Solo hará falta que ambos procesos tengan acceso al mismo sistema de intercambio de mensajes y que este permita identificar de alguna manera el emisor y el receptor. Hay que darse cuenta que se trata identificación interna del sistema de mensajería. En ningún caso son referencias directos a los procesos.

En la **comunicación indirecta** se desconocen las referencias a los procesos emisor y receptor del mensaje. Es necesaria la existencia de un sistema de mensajería capaz de distribuir mensajes identificados con algún valor específico y único, y permitir a los procesos la consulta de los mensajes a partir de este identificador para discernir los que vayan dirigidos a ellos. En cierta forma es como vehicular la comunicación a través de un buzón común (el identificador). El emisor envía el mensaje a un buzón específico y el receptor recoge el mensaje del mismo buzón.

Las órdenes serían de la siguiente forma:
enviar(Buzon_A,mensaje) → El proceso emisor deja el mensaje en el buzón A
recibir(Buzon_A,mensaje) → El proceso receptor recoge el mensaje del buzón A

Este sistema es más flexible que la comunicación directa puesto que nos permite tener más de un enlace de comunicaciones entre los dos procesos y podemos crear enlaces de tipos uno a muchos, muchos a uno y muchos a muchos. En los enlaces de comunicación uno a muchos como por ejemplo los sistemas cliente/servidor los buzones se denominan **puertos**.

Estos múltiples enlaces no se podrían hacer con el sistema de comunicación directa. El sistema operativo es el encargado de realizar la asociación de los buzones a los procesos y lo puede hacer de forma dinámica o estática.

Por otro lado, si tenemos en cuenta la inmediatez en la recepción de los mensajes entre procesos hablaremos de: comunicación **síncrona** o **asíncrona**.

En la **comunicación síncrona** el proceso que envía un mensaje a otro proceso se queda a la espera de recibir respuesta. Por lo tanto, el emisor queda bloqueado hasta que recibe esta respuesta. Y viceversa. En cambio en la **comunicación asíncrona** el proceso que envía el mensaje continúa su ejecución sin preocuparse de si el mensaje ha sido recibido o no. Por lo tanto, aquí tendrá que existir una memoria intermedia o que se vayan acumulando los mensajes. El problema vendrá cuando la memoria intermedia se encuentre llena. El proceso que envía el mensaje queda bloqueado hasta que encuentra el espacio dentro de la memoria intermedia.

Un símil a estos dos tipos de sincronización podría ser el envío de mensajes a través de correo electrónico o de chat. El primero, que va dejando los mensajes a un buzón sin preocuparse si los receptor lo recibe o no, sería una comunicación asíncrona. En cambio en un chat la comunicación es totalmente síncrona puesto que el emisor y el receptor tienen que coincidir en el tiempo.

Implementación usando Java

La comunicación y sincronización entre procesos de ordenadores diferentes conectados en red (sistemas distribuidos) requieren de una mecanismo de transferencia que asuma la comunicación indirecta, los sockets. Se trata de unas bibliotecas que disponen de todos los mecanismos necesarios para enviar y recibir mensajes a través de la red.

Dentro del mismo ordenador, Java dispone de métodos como **wait()** , **notify()** y **notifyAll()**, que modifican el estado de un hilo parando o activando la ejecución de los procesos referenciados. Estos métodos tienen que ser invocados siempre dentro de un bloque sincronizado (con el modificador **synchronize**).

- **wait()** : pone el hilo que se está ejecutando en estado de espera. Este hilo abandona la zona de exclusión mutua y espera, en una cola de espera, a ser vuelto a a activar por un método **notify()** o **notifyAll()** .

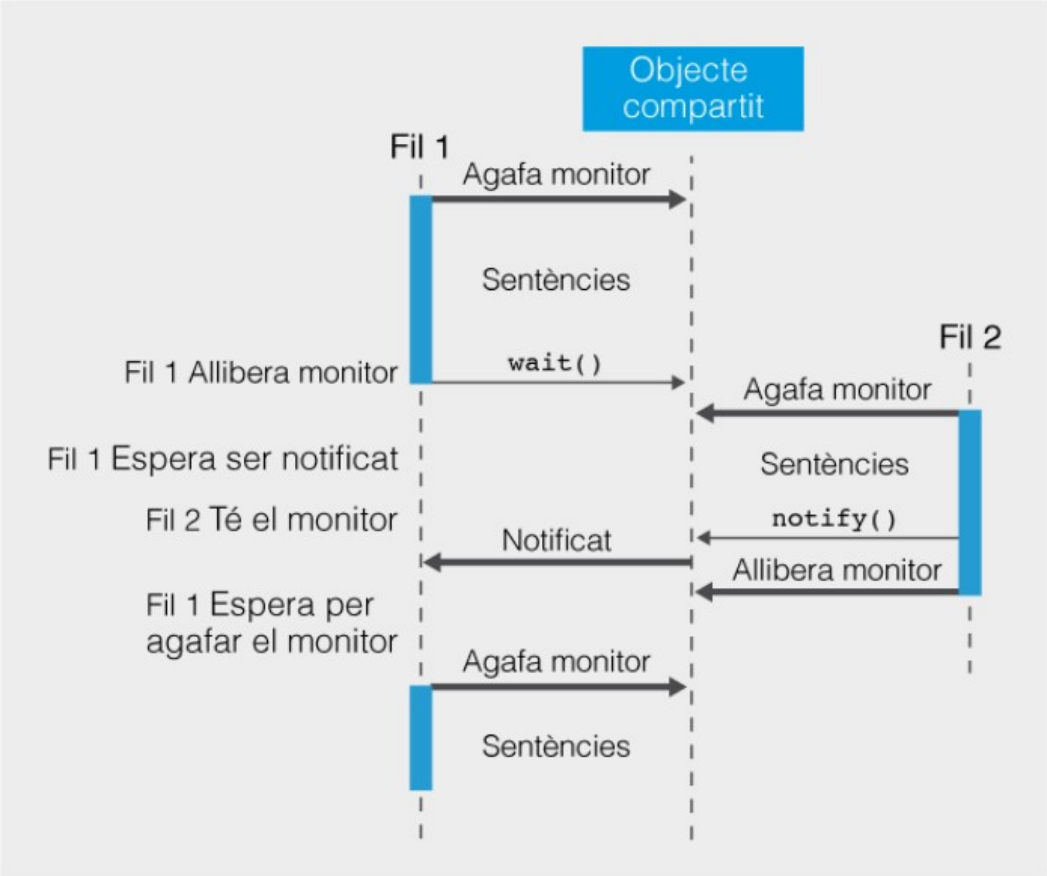
- **notify()** : un hilo de la cola de espera pasa en el estado de preparado.

- **notifyAll()** : todos los hilos de la cola de espera pasan en el estado de preparado.

Cuando un proceso entra en la sección crítica, es decir que controla el monitor, ninguno otro hilo puede entrar a la sección crítica del mismo objeto que está controlado por el mismo monitor. Para poder coordinarse y comunicarse se harán uso del métodos anteriores.

Cuando no se dan las condiciones para continuar con la ejecución de un hilo que se encuentra a la sección crítica, este puede abandonar la sección y permitir que otro hilo que se encuentra en espera pueda coger el monitor y entrar a la sección crítica. En la siguiente figura (figura 1.15) se muestra como dos hilos quieren acceder a un recurso común y se comunican para sincronizarse. Imaginamos que el recurso compartido es una tarjeta SIM de un teléfono móvil.

FIGURA 1.15. wait() i notify()



El hilo 1 quiere acceder a la tarjeta para hacer una llamada, coge el monitor del recurso compartido para tener acceso exclusivo a la tarjeta, pero la SIM no está activada todavía. La condición para poder ejecutarse no se cumple. Por lo tanto, ejecuta el método **wait()** que lo mantiene en espera hasta que la condición se cumpla. Cualquier otro proceso que quiera hacer la llamada y coja el monitor hará el mismo, ejecutará el método **wait()** y se quedará a la cola de espera.

El hilo 2 es un proceso que se encarga de activar la tarjeta SIM. Cuando coge el monitor tiene acceso con exclusión mutua a la tarjeta. Hace las sentencias para activar la SIM y ejecuta **notify()** . Esta sentencia notifica al primer hilo, que está a la cola de procesos en espera, que ya se cumplen las condiciones para poder continuar la ejecución de los procesos de llamadas y libera el monitor. La diferencia entre **notify()** y **notifyAll()** es que la notificación se hace a un hilo en espera o a todos. Por último, el hilo 1 coge el monitor y ejecuta la suyas sentencias de llamada.

En Java, haciendo uso de **synchronized** se crean **semáforos**. Si añadimos los métodos **wait()** , **notify()** o **notifyAll()** crearemos **monitores** fácilmente y de forma eficiente.