Multiprocesamiento y programación paralela

La informática del día a día requiere cada vez más potencia de cálculo en aplicaciones informáticas, cada vez más cálculos numéricos en ingeniería, ciencia, astrofísica, meteorología... Queremos que las computadoras reaccionen con el razonamiento humano. Así que tenemos ordenadores que juegan partidas de ajedrez, hablan como humanos o toman decisiones basadas en datos inexactos...

Aunque no es el único factor determinante, los procesadores juegan un papel fundamental en la consecución de esta potencia. Son cada vez más rápidos y eficientes y los sistemas operativos permiten coordinar varios procesadores para aumentar el número de cálculos por unidad de tiempo. El uso de varios procesadores dentro de un sistema informático se denomina multiprocesamiento. Sin embargo, la coordinación de varios procesadores puede provocar situaciones de error que tendremos que evitar.

Programación multiprocesamiento

Un chef que pretende hacer un plato muy elaborado, con diferentes salsas y guarniciones, probablemente utilizará diferentes paellas para cocinar cada elemento que necesite utilizar para hacer el plato. Tendrá una paella con la salsa, otra preparando la guarnición y en otra cocinará el plato principal. Todo esto lo hará al mismo tiempo y, finalmente, cuando todas estas tareas estén terminadas, se juntarán en un solo plato.

Esta analogía sirve para introducir el término **multiproceso.** El chef está ejecutando diferentes procesos al mismo tiempo: cocinar la salsa, la guarnición y el plato principal, siguiendo una orden de ejecución para llegar a un resultado que esperaba, un buen plato.

Sistemas de proceso, de un solo procesador y multiprocesador

Siguiendo con la analogía del chef, un proceso sería la actividad que realiza el chef (procesador) para elaborar uno de los complementos o el plato principal (resultados) a partir de una receta (programa). Cada proceso de creación requiere tomar los ingredientes (los datos), ponerlos en la sartén (recurso asociado) y cocinarlos independientemente del resto de los procesos.

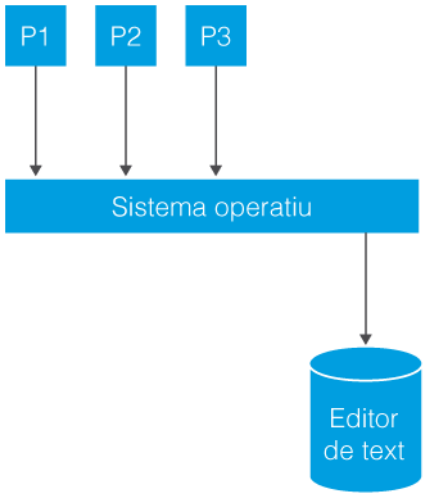
Un programa es un elemento estático, un conjunto de instrucciones, líneas de código escritas en un lenguaje de programación, que describen el tratamiento que se debe dar a los datos iniciales (entrada) para lograr el resultado esperado (una salida concreta). En cambio, **un proceso** es dinámico, es una instancia de un programa en ejecución, que realiza los cambios indicados por el programa a los datos iniciales y obtiene una salida específica. El proceso, además de las instrucciones, también requerirá recursos específicos de ejecución, como el contador de instrucciones del programa, el contenido de los registros o los datos.

El sistema operativo es responsable de la gestión de los procesos.

El sistema operativo se encarga de la gestión de procesos. Los crea, los elimina y les proporciona instrumentos que permiten su ejecución y también la comunicación entre ellos. Cuando se ejecuta un programa, el sistema operativo crea una instancia del programa: el proceso. Si el

programa se ejecutara de nuevo, se crearía una nueva instancia completamente independiente de la instancia anterior (un nuevo proceso) con sus propias variables, pilas y registros.

Imagina un servidor de aplicaciones en el que hemos instalado un programa de edición de texto. Digamos que hay varios usuarios que quieren escribir sus textos ejecutando el editor. Cada instancia del programa es un proceso totalmente independiente de los demás. Cada proceso tiene datos de entrada y, por lo tanto, diferentes salidas. Cada usuario escribirá a su ejecución del editor (el proceso), su texto. La figura .1 refleja este ejemplo.



FiguraEjecución de procesos

Cuando un proceso se está ejecutando, está completamente en memoria y tiene asignados los recursos que necesita. Un proceso no puede escribir en zonas de memoria asignadas a otros procesos, la memoria no se comparte. Cada proceso tiene una estructura de datos en la que se guarda la información asociada a la ejecución del proceso. Esta área se denomina Bloque de Control de Procesos (BCP). Los procesos compiten con otros procesos ejecutados al mismo tiempo por los recursos del sistema. Opcionalmente, el sistema operativo también permite que los procesos se comuniquen y colaboren entre sí.

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

Un **sistema monoprocesador** es aquel que consiste solo en un procesador.

Un **sistema multiprocesador consta** de más de un procesador.

- Placa base multiprocesador

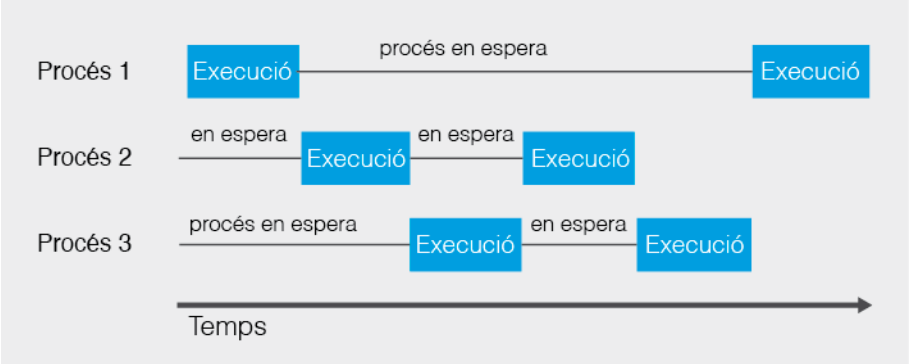
Actualmente, la mayoría de los sistemas operativos aprovechan los tiempos de descanso de los procesos, cuando esperan, por ejemplo, algunos datos de usuario o están pendientes de alguna operación de entrada/salida, para introducir otro proceso en el procesador, simulando así una ejecución paralela.

De forma genérica llamaremos a los procesos que se ejecutan al mismo tiempo, ya sea en procesos reales o simulados, **concurrentes.** La ejecución de procesos concurrentes, aunque sea simulada, aumenta el rendimiento del sistema informático ya que aprovecha más el tiempo del procesador. Es el sistema operativo encargado de gestionar la ejecución concurrente de diferentes procesos contra un mismo procesador y muchas veces nos referimos a él como **multiprogramación.**

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

Lafigura .2 corresponde a una imagen de procesos concurrentes en la que hay tres procesos en curso y se intercala el tiempo del procesador.

Figura Ejecución de procesos simultáneos



Procesador Intel con dos núcleos

El sistema operativo Windows 95 sólo le permite trabajar con un solo procesador. Los sistemas operativos de Win2000/NT y Linux/Unix son multiprocesamiento, pueden funcionar con más de un procesador

Los sistemas informáticos de monoprocesamiento actuales tienen una gran velocidad y si tienen que ejecutar más de un proceso a la vez alternarán su ejecución simulando un multiproceso. Cuando las técnicas multiprocesador se aplican a una computadora monoprocesador, los beneficios en el rendimiento no son tan evidentes como en los sistemas informáticos multiprocesador.

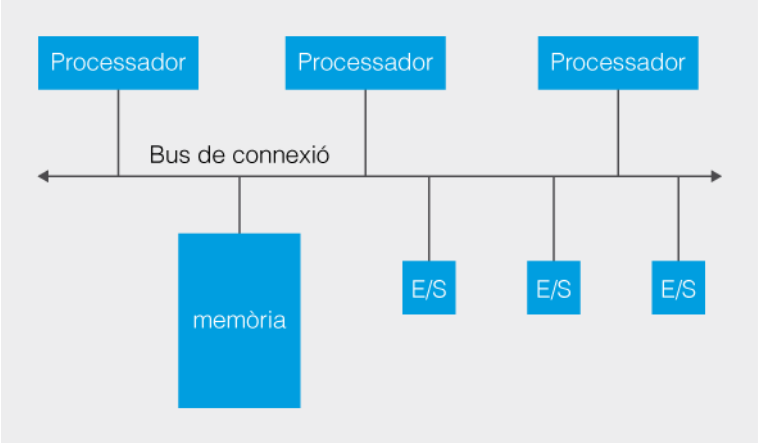
En un sistema informático multiprocesador hay dos o más procesadores, por lo tanto, se pueden ejecutar varios procesos simultáneamente. Los multiprocesadores también se pueden describir como aquellos dispositivos cuyo procesador tiene más de un kernel(multinúcleo),es decir, tienen más de una CPU en el mismo circuito de procesador integrado. Obviamente la arquitectura es diferente, pero este sistema, al igual que un equipo multiprocesador, nos permite ejecutar simultáneamente diferentes procesos.

Los sistemas multiprocesador se pueden clasificar dependiendo de su arquitectura en sistemas multiprocesador fuertemente acoplados y sistemas multiprocesador débilmente acoplados.

Sistemas multiprocesador fuertemente acoplados:en esta arquitectura los diferentes procesadores comparten la misma memoria y están interconectados a ella a través de un bus. También pueden tener una pequeña caché en cada procesador. Existe una fuerte colaboración entre los procesadores que comparten variables en la memoria común como resultados parciales o también para comunicarse entre sí. Actualmente, la mayoría de los sistemas operativos admiten este tipo de sistema multiprocesador.

Los sistemas fuertemente acoplados dependen del peso de cada procesador al ejecutar procesos. Se pueden dividir en sistemas multiprocesamiento simétricos,en los que los procesadores del sistema son de características similares y compiten con sus pares para ejecutar procesos, y sistemas de multiprocesamiento asimétricos en los que uno de los procesadores del sistema, el maestro,controla el resto de los procesadores. Este sistema también se llama amo-esclavo. muestra gráficamente cómo es un sistema multiprocesador fuertemente acoplado.

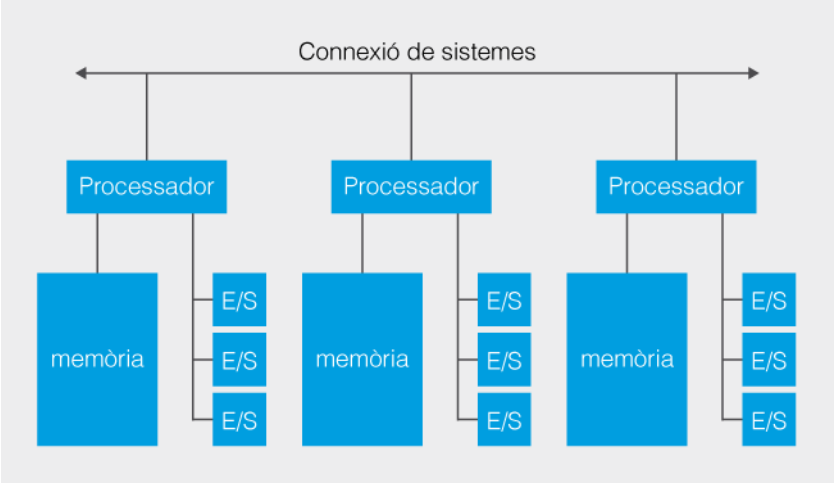
Figura Sistema informático multiprocesador fuertemente acoplado



Sistemas multiprocesamiento débilmente acoplados estos sistemas no comparten memoria, cada procesador tiene una memoria asociada. Las ejecuciones que requieren colaboración entre procesos son necesarias para intercambiar mensajes a través de enlaces de comunicación, para permitir la comunicación entre procesos. Un tipo de estos sistemas mal ensamblados son los sistemas distribuidos, en los que cada dispositivo monoprocesador (o multiprocesador) podría ubicarse en lugares físicamente distantes. Una red de computadoras o Internet podría ser un ejemplo de un sistema distribuido débilmente acoplado.

La siguiente imagen muestra gráficamente cómo es un sistema multiprocesador débilmente acoplado.

Figura Sistema informático multiprocesador débilmente acoplado



Programación concurrente, paralela y distribuida

Simula, creado en 1966, fue el primer lenguaje de programación concurrente orientado a objetos.

Actualmente, un gran número de aplicaciones utilizan programación concurrente. En los sistemas operativos actuales hay muchas aplicaciones que se ejecutan al mismo tiempo y comparten información. Por ejemplo, podemos escribir en un editor de texto, tener un reproductor de música encendido y descargar un vídeo al mismo tiempo. También podemos tener un navegador funcionando capaz de cargar en varias pestañas diferentes páginas web. Estos son ejemplos que ilustran la idea de programación concurrente.

- Azul profundo. Superordenador IBM con 256 procesadores trabajando codo con codo

Gracias a la evolución que ha experimentado el hardware en los últimos años, se han creado sistemas operativos que pueden optimizar los recursos del procesador y pueden ejecutar diferentes procesos simultáneamente. La ejecución simultánea de procesos también se denomina

concurrencia. No significa que tengan que ejecutarse exactamente al mismo tiempo, los procesos intercalados también se consideran ejecución concurrente. La mayoría de los lenguajes de programación actuales pueden hacer uso de este recurso y diseñar aplicaciones en las que los procesos se ejecutan simultáneamente.

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

La ejecución concurrente puede dar lugar a ciertos problemas a la hora de acceder a los datos que nunca encontraremos en la ejecución secuencial y que hay que tener en cuenta. Básicamente utilizaremos dos técnicas para evitarlos: el bloqueo y la comunicación.

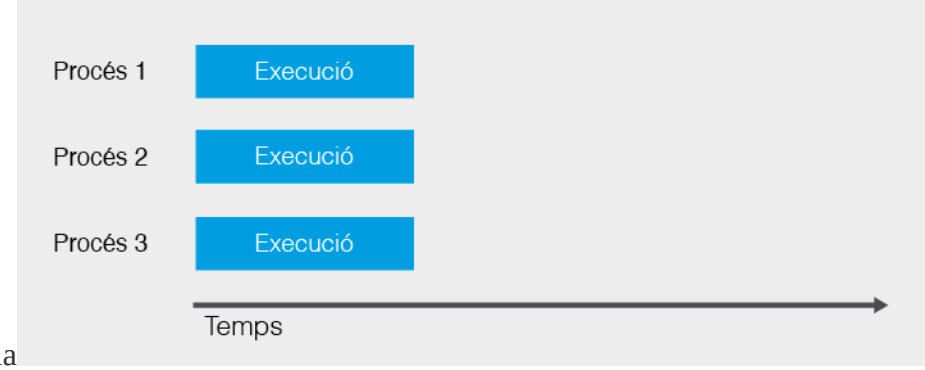
Si la programación concurrente se produce en un ordenador con un solo procesador hablaremos de multiprogramación. Por otro lado, si el ordenador tiene más de un procesador y, por tanto, los procesos se pueden ejecutar de forma realmente simultánea, hablaremos de programación paralela.

En un dispositivo multiprocesador la concurrencia es real, los procesos se ejecutan simultáneamente en diferentes procesadores del sistema. Es común que el número de procesos que se ejecutan simultáneamente sea mayor que el número de procesadores. Por lo tanto, será obligatorio que algunos procesos se ejecuten en el mismo procesador.

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

En los ordenadores multiprocesamiento la concurrencia es real, por lo que el tiempo de ejecución suele ser menor que en los dispositivos de monoprocesamiento.

En la figura.5 vemos un esquema del resultado de una ejecución de tres procesos en paralelo en una computadora multiprocesador con tres procesadores. En contraste tenemos la figura.2 con la ejecución en un ordenador monoprocesador.



FiguraEjecución paralela

La principal desventaja de la programación paralela son los controles que debemos añadir para que la comunicación y sincronización de los procesos que se ejecutan simultáneamente sean correctos. Dos procesos deben sincronizarse o comunicarse cuando un proceso necesita algunos

datos que está procesando el otro o uno de ellos debe esperar el final del otro para continuar su ejecución.

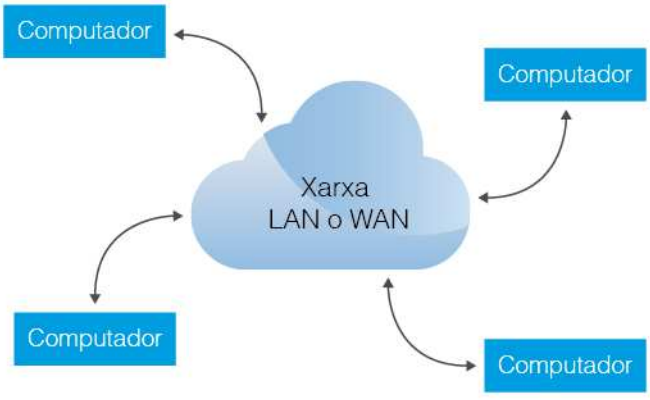
El lenguaje de programación Java ha integrado el soporte a la concurrencia en el propio lenguaje y no a través de librerías.

La programación paralela está estrechamente ligada a la arquitectura del sistema informático. Pero cuando programamos en idiomas de alto nivel tenemos que abstenernos de la plataforma en la que se ejecutará. Los lenguajes de alto nivel nos proporcionan librerías que facilitan esta abstracción y nos permiten utilizar las mismas operaciones para programar en paralelo que acabarán implementadas para utilizar una plataforma específica.

Un tipo especial de programación paralela es la llamada programación distribuida. Esta programación se produce en sistemas informáticos distribuidos. Un sistema distribuido consiste en un conjunto de ordenadores que pueden ubicarse en diferentes lugares geográficos vinculados entre sí a través de una red de comunicaciones. Un ejemplo de un sistema distribuido puede ser el de un banco con muchas sucursales en el mundo, con un ordenador central por oficina para almacenar cuentas locales y realizar transacciones locales. Este equipo puede comunicarse con otros equipos centrales de la red de la oficina. Cuando se realiza una transacción, no importa dónde se encuentre la cuenta o el cliente.

En la figura.6 podemos ver la ejecución de una programación distribuida.

FiguraEjecución de programación distribuida



La programación distribuida es un tipo de programación concurrente en la que los procesos se ejecutan en una red de procesadores autónomos o en un sistema distribuido. Es un sistema de ordenadores independientes que desde el punto de vista del usuario del sistema se ve como un único ordenador.

En los sistemas distribuidos, a diferencia de los sistemas paralelos, no existe memoria compartida, por lo que si necesitan el uso de variables compartidas, se crearán réplicas de estas variables en los diferentes dispositivos de la red y habrá que controlar la consistencia de los datos. La comunicación entre procesos para intercambiar datos o sincronizarse entre sí se realiza con mensajes que se envían a través de la red de comunicaciones que comparten.

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

Ventajas y desventajas del multiprocesamiento

Hemos definido la programación concurrente o concurrencia como la técnica mediante la cual se ejecutan múltiples procesos al mismo tiempo y pueden comunicarse entre sí. La mayoría de los sistemas intentan aprovechar esta concurrencia para aumentar la capacidad de ejecución.

Aumentar la potencia de cálculo y el rendimiento es una de las principales ventajas. Cuando se ejecutan varios procesos al mismo tiempo, se puede aumentar la velocidad de ejecución general. Sin embargo, debe tenerse en cuenta que no siempre es así. A veces, dependiendo de la complejidad de la aplicación, el sincronismo o las técnicas de comunicación son más costosas en el tiempo que la ejecución de procesos.

Un sistema multiprocesador es **flexible,**ya que si aumentan el proceso que se está ejecutando es capaz de distribuir la carga de trabajo de los procesadores, y también puede reasignar dinámicamente recursos de memoria y dispositivos para ser más eficiente. Es fácil de **cultivar.** Si el sistema lo permite, se pueden añadir nuevos procesadores fácilmente y así aumentar su potencia.

Los sistemas multiprocesador pueden ser sistemas redundantes. Disponer de varios procesadores puede permitir un aumento en la disponibilidad de recursos (más procesadores al servicio del usuario) o el uso de procesadores especializados en tareas de verificación y control. En el último caso hablaremos de sistemas con una alta **tolerancia a fallos.** Un error del procesador no detiene el sistema.

Los sistemas multiprocesamiento nos permiten diferenciar procesos por su **especialización** y, por tanto, reservar procesadores para operaciones complejas, aprovechar otros para el procesamiento paralelo y avanzar en la ejecución.

Las desventajas del multiprocesamiento se deben principalmente al control que se debe realizar cuando hay varios procesos en marcha y se debe compartir información o comunicarse. Esto aumenta la complejidad de la programación y penaliza el tiempo de ejecución.

Si el multiprocesamiento se encuentra en un entorno multiprocesador paralelo o distribuido, el tráfico de los buses de comunicación se incrementa paralelamente al número de procesadores u ordenadores que incorporamos al sistema, y esto puede convertirse en un cuello de botella crítico.

Procesos y servicios

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

Un servicio es un tipo de proceso que no tiene una interfaz de usuario. Pueden ser, dependiendo de su configuración, inicializados por el sistema de forma automática, en los que realizan sus funciones sin que el usuario lo sepa o pueden mantenerse a la espera de que alguien realice una solicitud para hacer una tarea concreta.

Dependiendo de cómo se estén ejecutando, los procesos se clasificarán como procesosen primer planoy en segundoplano. Los procesos en primer plano mantienen una comunicación con el usuario, ya sea informando de las acciones realizadas o esperando sus solicitudes a través de una interfaz de usuario. Por otro lado, los que se ejecutan en segundo plano no se muestran explícitamente al usuario, ya sea porque no necesitan su intervención o porque los datos requeridos no se incorporan a través de una interfaz de usuario.

El nombre deamon proviene del acrónimo inglés DAEMON(Disk And Execution Monitor). A menudo en la literatura técnica este concepto se ha extendido usando la palabra "demonio", por lo que puede encontrar esta palabra refiriéndose a los programas que se ejecutan en segundo plano.

Los servicios son procesos ejecutados en segundo plano. Service es una nomenclatura utilizada en Windows. En los sistemas Linux también se les llama procesos deamon.

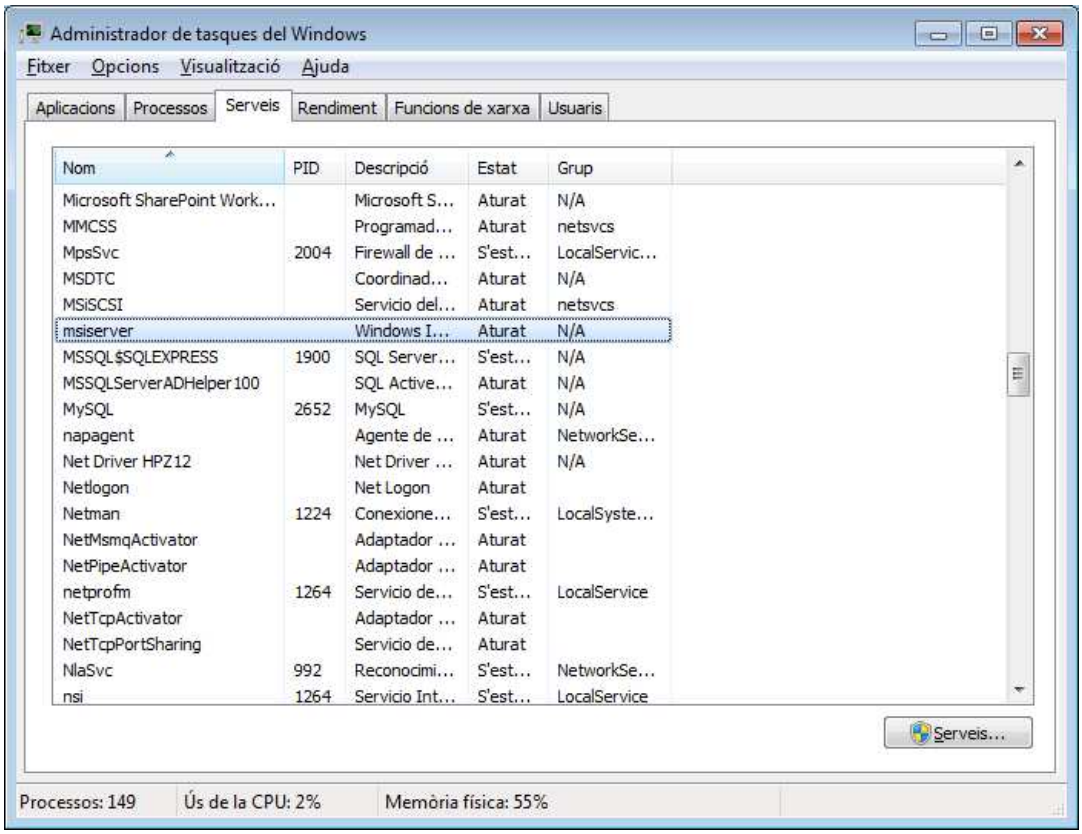
Dado que los servicios no cuentan con una interfaz de usuario directa, almacenan la información generada o los posibles errores que están produciendo, en archivos de registro generalmente conocidos como logs.

En Linux, el nombre de los demonios siempre termina con la letra d,como httpd, el demonio de http.

Algunos servicios pueden estar esperando ser llamados para realizar sus tareas. Por ejemplo, un servidor Web tiene un servicio activo. En Linux es un httpd que está escuchando un puerto de red y cuando el usuario realiza una solicitud a través de la red para obtener una página web, es el demonio el que se encarga de tramitar la solicitud y enviar la página de devolución, si tiene acceso autorizado y la página existe, o enviar el mensaje informativo a través de la red indicando la imposibilidad de entrega.

En la figura.7 podemos ver los servicios ejecutados en un ordenador Windows.

FiguraServicios de Windows



Los servicios pueden estar ejecutándose localmente, en nuestro equipo, como Firewall, antivirus, conexión Wi-Fi, o los procesos que controlan nuestro ordenador, o pueden ejecutarse en un ordenador de un sistema informático distribuido o en Internet, etc. Las llamadas a servicios distribuidos se realizan en protocolos de comunicación. Por ejemplo, los servicios http, DNS (Domain Name System), FTP (File Transfer Protocol) son servicios ofrecidos por servidores que se encuentran en Internet.

Hilos y Procesos

Recuerda al chef que trabaja simultáneamente preparando un plato para diferentes comensales. El procesador de un sistema informático (cook) está ejecutando diferentes instancias del mismo programa (la receta). Además, el chef está haciendo diferentes partes del plato. Si la receta es salchicha con patatas, dividirá su tarea preparando la salchicha por un lado y las patatas por el otro. Ha dividido el proceso en dos subprocesos. Cada uno de estos subprocesos se denomina **subprocesos.**

El sistema operativo puede mantener varios procesos en ejecución al mismo tiempo mediante concurrencia o paralelismo. Además, dentro de cada proceso se pueden ejecutar varios hilos, por lo que se pueden ejecutar bloques de instrucciones que presentan cierta independencia al mismo tiempo. Los subprocesos comparten recursos de proceso (datos, código, memoria, etc.). En consecuencia, los hilos, a diferencia de los procesos, comparten memoria y si un subproceso modifica una variable en el proceso, el resto de subprocesos podrán ver la modificación cuando accedan a la variable.

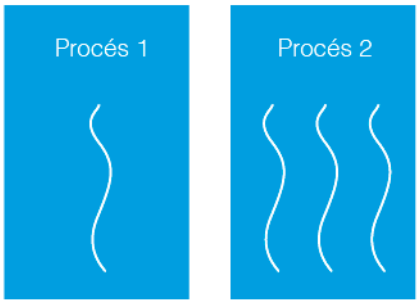
Los procesos se denominan **entidades pesadas** porque se encuentran en espacios de dirección de memoria independiente, creación y comunicación entre procesos, lo que consume muchos recursos del procesador. Por otro lado, ni la creación de subprocesos ni la comunicación consumen mucho tiempo de procesador. Por esta razón, los hilos se llaman **entidades de luz.**

Un proceso se ejecutará mientras uno de sus subprocesos esté activo. Sin embargo, también es cierto que si terminamos un proceso de forma forzada, sus hilos también terminarán la ejecución.

Podemos hablar de dos niveles de hilos: hilos a nivel de usuario, que son los que creamos al programar con un lenguaje de programación como Java; y subprocesos de segundo nivel que son los creados por el sistema operativo para admitir el primero. Programaremos nuestros hilos utilizando librerías que nos proporcionen el lenguaje de programación. Son las librerías con las instrucciones pertinentes que indicarán al sistema los hilos que tienen que crear y cómo gestionarlos.

La figura .8 muestra dos procesos, uno con la ejecución de un subproceso y el otro con la ejecución de tres subprocesos simultáneos.

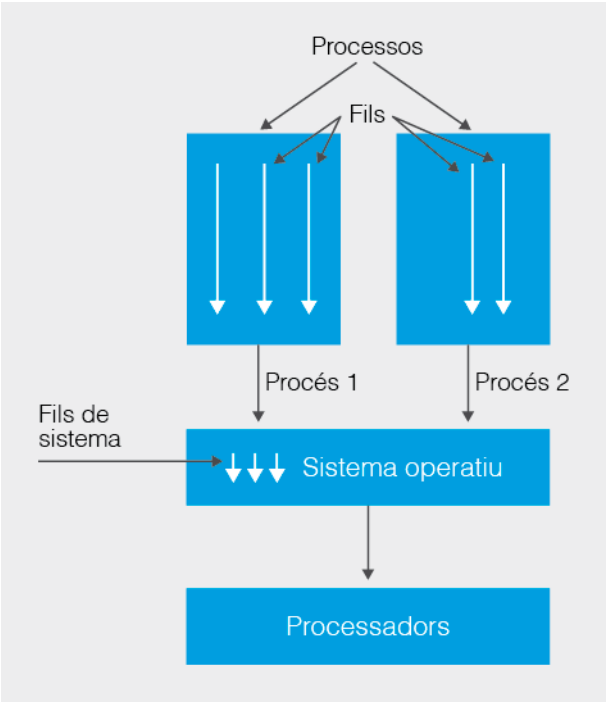
Figura 1.8 Proceso con un solo subproceso en ejecución y proceso con varios subprocesos en



ejecución

ilustra el tratamiento de los subprocesos por parte del sistema operativo. En cuanto al procesamiento, cada hilo se procesa de forma independiente incluso si pertenece o no al mismo proceso. La única diferencia entre el tratamiento de hilos y procesos se puede encontrar a nivel de memoria. Para subprocesos del mismo proceso, el sistema operativo debe mantener los mismos datos en la memoria. Para subprocesos de diferentes procesos en cambio es necesario tener datos independientes. En el caso de que sea necesario que el mismo procesador altere la ejecución de varios procesos, la memoria específica del proceso debe restaurarse en cada cambio. Esto también se llama cambio de contexto. Si la alternancia de procesamiento se realiza entre hilos del mismo proceso, no será necesario restaurar la memoria y por lo tanto el proceso será mucho más eficiente.

Figura 1.9 Dos procesos con diferentes subprocesos en ejecución. Cada subproceso se ejecuta de forma independiente. Los subprocesos del mismo proceso comparten memoria.



Gestión de procesos y desarrollo de aplicaciones para fines de computación paralela

Uno de los objetivos del sistema operativo es dotar a los procesos de los recursos necesarios para su ejecución. Debe lograr una directiva de programación que determine qué procesador se utilizará. Por lo tanto, el sistema operativo es responsable de decidir qué proceso ejecutar y durante cuánto tiempo es necesario hacerlo. Afortunadamente, los procesos contemplan muchas veces de descanso en las que no necesitan utilizar el procesador, por ejemplo esperando datos. Si en el inicio el proceso alerta al sistema operativo, puede reemplazar el proceso activo por otro en la disposición para usar el procesador. Al final del tiempo de descanso, los procesos deben notificar de nuevo al sistema para que planee una ejecución futura cuando el procesador esté libre. Cuando los procesos finalmente completen la ejecución, el sistema operativo liberará los recursos utilizados para la ejecución y liberará al procesador. El tiempo desde que un proceso comienza a ejecutarse hasta que finaliza se denomina ciclo de vida del proceso. Para que el sistema gestione las necesidades de los procesos a lo largo de su ciclo de vida, se identificarán etiquetándolos con un estado que indique sus necesidades de procesamiento.

Estados de un proceso

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

Dependiendo del sistema operativo y de si se está ejecutando en un proceso o subproceso, el número de estados puede variar. El diagrama de estado de la figura.10 muestra los estados más comunes.

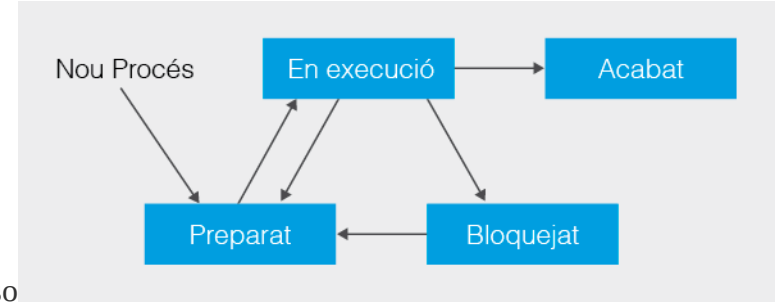


Figura Estados de un proceso

El primer estado que encontramos es **nuevo,**es decir, cuando se crea un proceso. Una vez creado, pasa al estado **preparado.** El proceso está actualmente listo para hacer uso del procesador, está compitiendo por el recurso del procesador. El planificador de procesos del sistema operativo es el que decide cuándo entra el proceso a ejecutar. Cuando el proceso se está ejecutando, su estado se denomina **ejecución.** De nuevo, es el planificador el encargado de decidir cuándo dejar el procesador.

Asignar un tiempo de ejecución fijo a cada proceso y después de este tiempo cambiar el proceso de estado de ejecución al estado preparado que espera ser asignado de nuevo al procesador, puede ser una política de planificador de procesos para asignar los diferentes runarios.
Process Planner sigue una política que indica cuándo un proceso debe cambiar de estado, cuándo debe liberarse o entrar en el uso del procesador.

Cuando un proceso sale del procesador, porque el procesador ha decidido, cambiará al estado preparado y se quedará esperando a que el planificador asigne el turno para recuperar el procesador y volver al estado de ejecución.

Desde el estado de ejecución, un proceso puede ir al estado **bloqueado o en espera.** En este estado, el proceso estará esperando un evento, como una operación de entrada/salida, o esperando la finalización de otro proceso, etc. Cuando ocurra el evento esperado, el proceso volverá al estado preparado, guardando el turno con el resto de procesos preparados.

El último estado está **terminado.** Es un estado que se alcanza una vez que el proceso ha completado toda su ejecución y estará listo para que el sistema libere cuando pueda los recursos asociados.

Las transiciones en el estado de un proceso pueden deberse a cualquiera de las siguientes razones:

- De la ejecución al bloqueado: el proceso realiza alguna operación entrante y saliente o el proceso debe esperar a que otro proceso modifique cualquier información o libere los recursos que necesita.
- De ejecución a listo: El sistema operativo decide quitar un procesador de procesador porque ha pasado un tiempo excesivo usándolo y lo pasa al estado preparado. Asigne acceso de procesador a otro procesador.
- De listo para ejecutar: es el planificador de procesos, dependiendo de la política que practiques, la persona encargada de realizar este cambio.
- De bloqueado a listo: el recurso o los datos para los que se había bloqueado están disponibles o la operación de entrada y salida ha finalizado.
- De ejecución a finalización: el proceso finaliza sus operaciones ya sea otro proceso o el sistema operativo lo hace terminar.

Cada vez que un proceso cambia al estado de ejecución, se denomina cambio de **contexto,**porque el sistema operativo debe almacenar los resultados parciales obtenidos durante la ejecución del proceso saliente y debe cargar los resultados parciales de la última ejecución del proceso introduciendo los registros del procesador, garantizando la accesibilidad a las variables y al código que tiene para seguir ejecutándose.

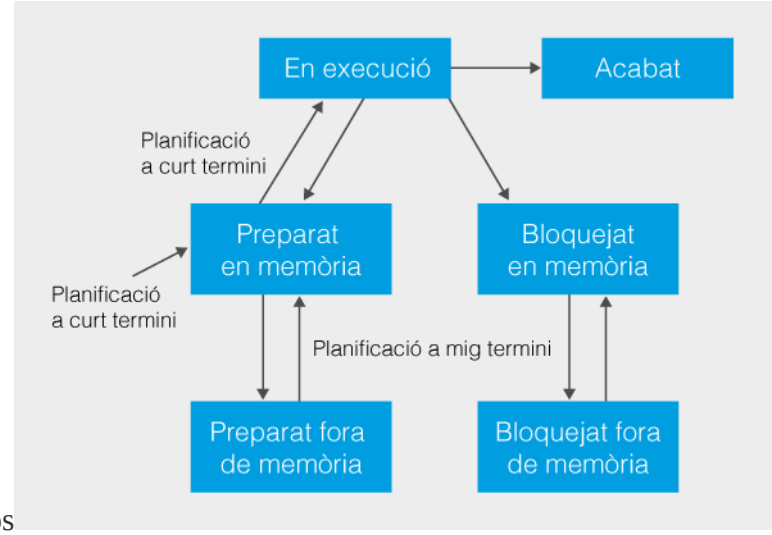
Programación de procesos

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

La función scheduler del núcleo del sistema operativo se encarga dedecidir qué proceso entra en el procesador, se puede dividir en tres niveles:

1. **Planificación a largo plazo:**En el momento en que se crea un proceso, se deciden algunos criterios de programación, como la unidad máxima de tiempo que un proceso puede permanecer en ejecución (llamada **quantum)**o la prioridad inicial que se asignará a cada proceso dinámicamente para el uso del procesador.
2. **Planificación a corto plazo:**cada vez que un proceso sale del procesador, es necesario tomar una decisión sobre cuál será el nuevo proceso que entrará en uso. En este caso, las políticas tratan de minimizar el tiempo necesario para lograr un cambio de contexto. Los procesos recién ejecutados generalmente tienen prioridad, pero es necesario asegurarse de que ningún proceso de ejecución se excluya permanentemente.
3. **Planificación a medio plazo:**hay otras partes del sistema operativo que también son importantes en la planificación de procesos. En el caso de la memoria SWAP, la eliminación de un proceso de la memoria debido a problemas de espacio significa que no se puede planificar de inmediato. El planificador a medio plazo será el encargado de decidir cuándo llevar los datos de los procesos almacenados fuera a la memoria principal y cuáles deberán transferirse de la memoria principal al espacio de intercambio.

La Figura.11 muestra gráficamente los niveles de programación de procesos.



FiguraProgramación de procesos

Los sistemas de multiprocesamiento también presentan dos componentes de planificación específicos.
El componente **de programación temporal** en el que se define individualmente la política de planificación que actúa sobre cada procesador, como si de un sistema monoprocesador se tratase. El planificador temporal decide cuánto tiempo lleva procesar cada proceso, cuándo cambiar, por qué razón, etc.
El componente **de programación espacial,**que define cómo se dividen los procesos entre los procesadores. Es decir, organizar qué procesador ejecuta qué procesador.
Esta programación en multiprocesadores utiliza algunos criterios para aplicar directivas de asignación. El planificador está memorizando cuáles son los procesos ejecutados en cada procesador y decide si será apropiado que se ejecute nuevamente. Una directiva**de afinidad de procesador** puede causar sobrecarga en algunos procesadores y hacer que la ejecución sea demasiado eficiente. Por el contrario, una política **de distribución** de carga evita este supuesto infrautilización de algunos procesadores, pero aumenta las sobrecargas en la memoria compartida.

Programación paralela transparente

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

Pero, ¿y si, en lugar de aplicar la concurrencia explícita, en la que el programador proporciona al algoritmo los patrones de paralelismo, se aplica una programación de concurrencia implícita, en la que es el sistema operativo el que decide qué procesos se pueden dividir y cuáles no, aplicando ciertas reglas genéricas y comunes a la mayoría de los problemas? El programador solo debe elegir las reglas.

Los sistemas informáticos multinucleares se han generalizado en servidores, computadoras de escritorio y también sistemas móviles, teléfonos inteligentes,tabletas, etc. Todos estos dispositivos ayudan a mejorar la programación concurrente y paralela. Pero la programación paralela es compleja, los procesos deben sincronizarse y compartirse el control de datos, lo que agrega complejidad a la tarea para el programador.

La plataforma Java en Java SE 6 introduce un conjunto de paquetes que proporcionan soporte para la programación concurrente y Java SE 7 mejora aún más el soporte para la programación paralela. Java proporciona soporte para programación concurrente con bibliotecas de bajo nivel a través de clases e interfaces. Sin embargo, hay problemas que se pueden programar en paralelo siguiendo patrones similares. El uso de i conlleva un esfuerzo adicional del programador, obligándole a añadir pruebas extra para verificar el correcto comportamiento del código, evitando lecturas y hechos erróneos, sincronizando las ejecuciones de los procesos y evitando los problemas derivados de los bloqueos. ***java.lang.Threadjava.lang.RunnableThreadRunnable***

Un enfoque relativamente simple para algunos de estos problemas que se pueden resolver en paralelo con implementaciones típicas que presentan un cierto nivel de creación de patrones son los

marcos predefinidos de programación paralela del lenguaje en Java: **Ejecutor** y **fork-join.** Son librerías que encapsulan gran parte de la funcionalidad y con ella la complejidad adicional que presenta la programación de menor nivel (). ***Threads***

Uso de patrones para resolver problemas similares

En ingeniería, llamamos patrones a aquellas soluciones genéricas que nos permiten resolver varios problemas similares cambiando cosas pequeñas. La ingeniería informática utiliza este concepto para referirse a bibliotecas complejas que permiten un alto grado de configuración aunque por tanto es necesario escribir aquellas partes concretas que adaptarán el patrón al problema que se está resolviendo.

Java utiliza este concepto en muchas de sus bibliotecas, definiendo una estructura base a través de interfaces y clases genéricas. El programador puede reutilizar esta estructura y el código ya implementado, usándolo en diferentes situaciones y adaptándolo con código específico para aquellas definiciones abstractas y clases que permanecen indefinidas en la implementación de la biblioteca.

La biblioteca es una de las que implementa diversos patrones con el objetivo de resolver de la manera más transparente posible gran parte de las situaciones en las que se necesita programación paralela. Cada patrón se adapta a diferentes situaciones por lo que es necesario saber distinguir situaciones y aplicar correctamente el patrón que sea necesario. ***Executor***

Ejecutor

Java nos da herramientas para la simplificación de la programación paralela. La idea es dividir un problema en problemas más pequeños y cada uno de estos subproblema los envía a ejecutarse por separado. Por lo tanto, tenemos varios más simples de una tarea complicada. Para ejecutar todas estas subtaskets en paralelo y así aumentar el rendimiento, se crean diferentes subprocesos o hilos, pero esta creación es totalmente transparente por parte del programador, es la máquina virtual Java la encargada de gestionar la creación de hilos para ejecutar los subtashs en paralelo.

Executor es una mejora en la creación de hilos, ya que se abstrae de la creación y gestión de subprocesos, de hecho solo es necesario indicar la tarea o tareas a realizar en paralelo (objetos instantáneos de tipo Runnable o Callable), elegir el gestor adecuado y dejar que se encargue de todo.

A grandes rasgos, podemos decir que tenemos tres tipos de gestores, uno genérico (), uno capaz de seguir pautas de ejecución temporal () y un último que por su especificidad y complejidad lo estudiaremos aparte. Es el llamado marco Fork-Join.

ThreadPoolExecutorScheduledThreadPoolExecutor

ThreadPoolExecutor, es la clase base que permite implementar el gestor genérico. Los administradores genéricos son adecuados para resolver problemas que se pueden resolver ejecutando tareas separadas entre sí. Imagina que queremos saber la suma de una gran lista de números. Podemos optar por añadir la lista de forma secuencial o dividirla en dos o tres piezas y añadir cada pieza por separado. Al final solo necesitas agregar el valor de cada trozo al resultado.

La suma de cada pieza es totalmente independiente y por lo tanto se puede hacer independientemente del resto. Es decir, solo sería necesario crear tantas tareas como bits para añadirlas y pasarlas al gestor para que las ejecute.

ThreadPoolExecutor, no crea un subproceso para cada tarea (instancia de o). De hecho, es común asignar más tareas que hilos. Las tareas pendientes permanecen en una cola de tareas y un cierto número de subprocesos son responsables de ejecutarlas. ¿Cuántos hilos necesitas crear? Estoy seguro de que esta es una decisión difícil y dependerá de cada caso. Es por ello que JAVA nos ofrece 3 formas rápidas de configurar instancias utilizando la clase. Es una clase que reúne un conjunto de utilidades en forma de métodos estáticos. Los tres tipos se obtienen a partir de los siguientes métodos estáticos: ***CallableRunnableThreadPoolExecutorExecutors***

- **newCachedThreadPool()**: crea un grupo de usuarios que crea subprocesos a medida que los necesita y reutiliza subprocesos inactivos.
- **newFixedThreadPool**(int numThreads): crea un grupo con un número de subproceso fijo. El indicado en el parámetro. Las tareas se almacenan en una cola de tareas y se asocian con subprocesos a medida que se vuelven libres.
- **newSingleThreadExecutor**(): crea un grupo con un único subproceso que procesará todas las tareas.

Cualquiera de estas configuraciones obtendrá una instancia que se comportará como se indica en los comentarios. Para agregar nuevas tareas, iniciar el procesamiento paralelo y mantenerlo activo hasta que se hayan procesado todas las tareas, la clase tiene métodos para ingresar una tarea (instancia de o) cada vez, para agregar, de repente, una lista de múltiples tareas, para obtener el número de subprocesos en ejecución o también para saber el número de tareas completadas. **ThreadPoolExecutorThreadPoolExecutorexecuteRunnableCallableinvokeAllgetPoolSize()getCompleteTaskCount()**
ScheduledThreadPoolExecutor es un caso concreto de pool. Se trata de un gestor de subprocesos, con una política de ejecución asociada a una secuencia de tiempo. Es decir, ejecutar las tareas programadas de vez en cuando. Puede ser una vez o repetitivamente. Por ejemplo si queremos revisar nuestro correo cada 5 minutos, mientras realizamos otros trabajos, podemos programar un hilo separado con esta tarea usando . La tarea podría revisar el correo y notificar solo si han llegado otros nuevos. De esta manera podemos concentrarnos en nuestro trabajo dejando que el hilo periódico trabaje para nosotros. Podemos obtener instancias de uso de una utilidad de la clase, el método estático.
ScheduledThreadPoolExecutorScheduledThreadPoolExecutorExecutornewScheduledThreadPool(int numThreads)

La clase tiene los siguientes métodos para administrar el retraso y la cadencia:
ScheduledThreadPoolExecutor

- schedule (Callable task, long delay, TimeUnit timeunit): crea y ejecuta una tarea(tarea)que se invoca después de un tiempo específico(retraso)expresado en las unidades indicadas(unidad de tiempo).
- schedule (Runnable task, long delay, TimeUnit timeunit): crea y ejecuta una acción(tarea),solo una vez, después de que haya transcurrido un tiempo específico(retraso)expresado en las unidades indicadas (unidad detiempo).

- scheduleAtFixedRate (Runnable task, long initialDelay, long period, TimeUnit timeunit): Crea y ejecuta una tarea(task)periódicamente. La primera vez se invocará después de un retraso inicial(initialDelay)y luego cada período de tiempo dado(período). El tiempo se mide en las unidades indicadas (unidad de tiempo).
- scheduleWithFixedDelay (Runnable task, long initialDelay, long delay, TimeUnit timeunit): Crea y ejecuta una tarea(task)periódicamente. La primera vez se invocará después de un retrasoinicial (initialDelay)y luego, al final de la última tarea lanzada, la siguiente tarea no comenzará, hasta que haya pasado el tiempo indicado en retraso. El tiempo se mide en las unidades indicadas (unidad de tiempo).

Los métodos que reciben tareas de tipo devolverán el resultado obtenido durante el cálculo. Callable

Ejemplo con ThreadPoolExecutor

Veamos ahora un ejemplo para calcular diez multiplicaciones enteras aleatorias usando una con un máximo de 3 hilos. Para el establecimiento del gerente será necesario: Executor

```
ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newFixedThreadPool(3);
```

Esto significa que si hay más de 3 tareas por ejecutar, solo enviará 3 para ejecutar y el resto se bloqueará hasta que un subproceso termine su procesamiento.

La clase interna implementa la interfaz. El método run podría usarse e implementarse. La diferencia es que devuelve valores de resultado en forma de cualquier objeto. La concurrencia se lleva a cabo dentro del método. MultiplicacioCallableRunableCallablecall()

Se crea una lista de diez multiplicaciones aleatorias. Y luego llama al método ejecutor. Recibe una colección de tareas de tipo, las ejecuta y devuelve el resultado dentro de un objeto. La clase es una interfaz diseñada para admitir clases que mantienen datos calculados de forma asincrónica. Las instancias requieren que utilice métodos para acceder a los resultados. Esto garantiza que el valor real solo será accesible cuando finalicen los cálculos. Mientras duren los cálculos la invocación del método implicará un bloqueo del hilo que lo está invocando.

invokeAll()CallableFuturejava.util.concurrent.FutureFuturegetget

Los objetos futuros se generan como resultado de la ejecución del método de instancias. Para cada objeto invocado se genera un objeto Future que permanecerá bloqueado hasta que finalice la instancia invocable asociada. callCallableCallable

```
package multiplicallista;

import java.util.*;
import java.util.concurrent.*;

public class MultiplicaLlista {

static class Multiplicacio implements Callable<Integer> {
 private int operador1;
 private int operador2;
 public Multiplicacio(int operador1, int operador2) {
   this.operador1 = operador1;
   this.operador2 = operador2;
 }
 @Override
 public Integer call() throws Exception {
    return operador1 * operador2;
 }
}
```

```
 public static void main(String[] args) throws InterruptedException, ExecutionException {

    ThreadPoolExecutor executor = (ThreadPoolExecutor) Executors.newFixedThreadPool(3);
    List<Multiplicacio> llistaTasques= new ArrayList<Multiplicacio>();
    for (int i = 0; i < 10; i++) {
      Multiplicacio calcula = new Multiplicacio((int)(Math.random()*10), (int)(Math.random()*10));
      llistaTasques.add(calcula);
    }
    List <Future<Integer>> llistaResultats;
    llistaResultats = executor.invokeAll(llistaTasques);

    executor.shutdown();
    for (int i = 0; i < llistaResultats.size(); i++) {
         Future<Integer> resultat = llistaResultats.get(i);
         try {
           System.out.println("Resultado de la tarea "+i+ " es:" + resultat.get());
         } catch (InterruptedException | ExecutionException e) {
         }
      }

  }
}
```

ThreadPoolExecutor también tiene el método que recibe una colección de instancias que ejecutará hasta que una de ellas finalice sin error. En este momento devolverá el resultado obtenido por la tarea completada en un objeto. invokeAny(tasques)CallableFuture

Otro método a conocer es , que le impide ejecutar nuevas tareas, pero permite que los subprocesos que se están ejecutando puedan terminar. ThreadPoolExecutorshutdown()

Ejemplo con ScheduledThreadPoolExecutor

En el siguiente ejemplo usaremos el ejecutor para programar tareas que se ejecuten periódicamente, después de un retraso programado. ScheduledThreadPoolExecutor

Primero creamos un pool de dos hilos del y uso, que nos permite programar y ejecutar el inicio de la tarea () y programar las tareas sucesivas tras la primera finalización(largo periodo). ScheduledThreadPoolExecutorscheduleWithFixedDelayinitialDelay

Ejecuta el método run de . La primera vez después de 2 segundos y luego cada 3 segundos. ExecutaFil

Finalmente, el método permanece en espera antes del apagado (detenido), hasta que todas las tareas hayan finalizado, se produzca el tiempo de espera (tiempo de espera) o los subprocesos finalicen abruptamente, lo que ocurra primero. awaitTermination

```
package tascaprogramada;

import java.util.Calendar;
import java.util.GregorianCalendar;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;

public class TascaProgramada {
```

```
   public static void main(final String... args) throws InterruptedException, ExecutionException {
      //mostramos la hora de la ejecución
       Calendar calendario = new GregorianCalendar();
      System.out.println("Inicio: "+ calendario.get(Calendar.HOUR_OF_DAY) + ":" +
calendario.get(Calendar.MINUTE) + ":" + calendario.get(Calendar.SECOND));

       // Crea un pool de 2 hijos
       final ScheduledExecutorService schExService = Executors.newScheduledThreadPool(2);
       // Crea el objeto Runnable.
       final Runnable ob = new TascaProgramada().new ExecutaFil();
       // Programa Hijo, se inicia a los dos segundos y después se va a ejecutar cada 3 seg
       schExService.scheduleWithFixedDelay(ob, 2, 3, TimeUnit.SECONDS);
       // Espera para acabar 10 segundos
       schExService.awaitTermination(10, TimeUnit.SECONDS);
       // shutdown .
       schExService.shutdownNow();
       System.out.println("Completado");
   }
   // Hijo Runnable
   class ExecutaFil implements Runnable {
      @Override
      public void run() {
         Calendar calendario = new GregorianCalendar();
         System.out.println("Hora de ejecución de la tarea: "+
calendario.get(Calendar.HOUR_OF_DAY) + ":" + calendario.get(Calendar.MINUTE) + ":" +
calendario.get(Calendar.SECOND));
         System.out.println("Tarea en ejecución");

         System.out.println("Ejecución acabada");
      }
   }


}
```

Cuando ejecutemos el código anterior obtendremos resultados similares a los siguientes en pantalla:

```
correr:
Inicio: 11:27:10
Tiempo de ejecución: 11:27:12
Ejecución de tareas
Ejecución finalizada
Tiempo de ejecución: 11:27:15
Ejecución de tareas
Ejecución finalizada
Tiempo de ejecución: 11:27:18
Ejecución de tareas
Ejecución finalizada
Completado
BUILD SUCCESSFUL (tiempo total: 10 segundos)
```

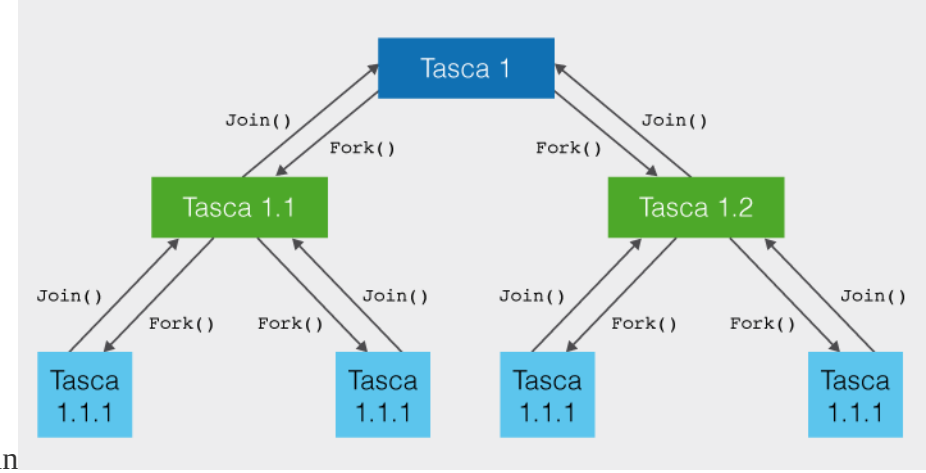
Observe cómo se produce la primera ejecución después de 2 segundos, luego la tarea se ejecuta cada 3 segundos.

Fork-Join

El marco Fork-Join es una evolución de los patrones de ejecutor. De hecho aquí también encontramos un gestor de hilos (la clase), pero un poco más sofisticado. Procese las tareas con el algoritmode robo de trabajo. Esto significa que el administrador del grupo busca subprocesos poco activos e intercambia tareas en espera. Además, las tareas pueden crear nuevas tareas que se agregarán a la cola de tareas pendientes que se realizarán. La eficiencia lograda es bastante alta, ya que el procesamiento paralelo se utiliza al máximo. Por este motivo, Fork_Join es una alternativa ideal para algoritmos que se pueden resolver de forma recursiva ya que se conseguirá la máxima eficiencia intercambiando aquellas tareas que están a la espera de resultados por otras que permitan avanzar en los cálculos. ForkJoinPool

La clase es una clase abstracta para tareas que se ejecutan en y contiene los métodos y . El método coloca la tarea invocada en la cola de ejecución en cualquier momento para que se programe. Esto permite que una tarea cree otras nuevas y las deje listas para ser ejecutadas cuando el gerente lo considere. ForkJoinTaskForkJoinPoolforkjoinfork()
El método impedirá que el subproceso invocador se ejecute en espera de la tarea invocada para completar la ejecución y devolver, si procede, los resultados. El bloqueo de subprocesos alertará al administrador de que puede cambiar la tarea detenida por otra que permanezca en espera. join()ForkJoinPool

La figura.12 muestra cómo las tareas cooperan a través de métodos y . fork()join()



FiguraMétodos fork-join

ForkJoinTask se implementa mediante dos clases que se especializan en ello, la clase y la clase. El primero es adecuado para tareas que necesitan calcular y devolver un valor. El segundo, por otro lado, representa procedimientos o acciones que no necesitan devolver ningún resultado. RecursiveTaskRecursiveAction

Cuando lo usamos es muy práctico utilizar el método, en lugar de llamar, para cada tarea generada los métodos y . Específicamente, el método invoca cada una de las tareas recibidas como parámetro y luego se espera la finalización de cada subproceso con una llamada al método de las tareas iniciadas. no devuelve ningún resultado, por lo que no es muy útil para usar con objetos de tipo. RecursiveActioninvokeAllforkjoininvokeAllforkjoininvokeAllRecursiveTask

Algoritmos recursivos y algoritmos iterativos

Los algoritmos recursivos tienen la peculiaridad de poder resolverse a sí mismos realizando la misma operación varias veces pero utilizando diferentes valores. Cada ejecución es responsable de realizar un cálculo parcial utilizando los datos de una o más ejecuciones anteriores de la misma operación. El último resultado parcial calculado coincidirá con la solución final. Para que el cálculo recursivo sea factible, es necesario que exista una solución trivial para que cualquiera de los valores funcione. Por ejemplo, el agoritm que calcula el valor factorial de un número es recursivo porque podemos decir que el valor factorial de un entero positivo es: n! = n* (n-1)! excepto cuando n vale 1 (el caso trivial) ya que sabemos que 1! = 1.

Usando un lenguaje de programación podemos expresar este algoritmo recursivo mediante:

```
long factoria(long n){
   if(n==1){
      return n;
   }else{
      return n*factorial(n-1);
   }
}
```

Cabe recordar que los algoritmos recursivos consumen mucha memoria y que en el procesamiento secuencial, pueden ser ineficientes si su cálculo requiere muchas llamadas recursivas. Pero cualquier algoritmo recursivo siempre se puede transformar en iterativo. En el procesamiento secuencial, los algoritmos iterativos son más eficientes que los algoritmos recursivos. Sin embargo, es posible aprovechar el procesamiento paralelo y las características recursivas de algunos algoritmos para gestionar llamadas recursivas en paralelo de manera que aumentemos la eficiencia final. Desafortunadamente, el aumento de la eficiencia solo será efectivo si el cálculo a realizar tiene suficiente entidad.

Para dar la entidad correcta a los cálculos y optimizar la eficiencia, el cálculo iterativo se combina con el cálculo recursivo. Es decir, se define un umbral o condición a partir del cual sería preferible resolver el problema iterativamente. Hasta que se alcance el umbral, el cálculo se derivará a la versión recursiva ejecutada en paralelo, utilizando el marco Fork-Join. Cuando lleguemos al umbral, por otro lado, realizaremos el cálculo secuencialmente. La elección del umbral no es trivial porque dependerá en gran medida de tener un algoritmo eficiente o no.

Por lo tanto, el esquema de ejecución de las tareas que se implementarán en las instancias de forkJoinTask sería similar al código siguiente:

```
if (para resolverlo sale más a cuenta resolverlo, resolver directamente ){ //
  ejecución secuencial
}else{
  división recursiva de la tarea.
  llamada/programación de ejecución paralela de las tareas creadas
  recopilación de resultados y combinación adecuada para obtener el calculo deseado.
}
```

Veamos lo que acabamos de explicar con un ejemplo. Veremos dentro de una matriz de enteros el mayor número. La recursión se basará en dividir la colección de enteros en dos y pedir por cada fragmento para obtener el valor mínimo y el cálculo secuencial en un algoritmo de búsqueda iterativo.
Usaremos RecursiveTask porque necesitamos devolver el valor mínimo. Los valores iniciales siempre se pasarán a través del constructor de tareas y los almacenarán como atributos para tenerlos disponibles en el método que es el equivalente específico de las tareas en o ya visto cuando se habla del . ComputeForkJoinTaskruncallExecutor

```
package ioc.dam.m09.u1.forkjoin.maxim;

import java.util.concurrent.RecursiveTask;
import java.util.concurrent.ForkJoinPool;

public class MaximTask extends RecursiveTask<Short> {
   //private static final int LLINDAR=10000000;
   private static final int LLINDAR=10000000;
   private short[] arr ;
   private int inici, fi;

   public MaximTask(short[] arr, int inici, int fi) {
         this.arr = arr;
         this.inici = inici;
         this.fi = fi;
   }

   private short getMaxSeq(){
      short max = arr[inici];
      for (int i = inici+1; i < fi; i++) {
         if (arr[i] > max) {
               max = arr[i];
         }
      }
      return max;
   }

   private short getMaxReq(){
      MaximTask task1;
      MaximTask task2;
      int mig = (inici+fi)/2+1;
      task1 = new MaximTask(arr, inici, mig);
      task1.fork();
      task2 = new MaximTask(arr, mig, fi);
      task2.fork();
      return (short) Math.max(task1.join(), task2.join());
   }

   @Override
   protected Short compute() {
      if(fi - inici <= LLINDAR){
         return getMaxSeq();
```

```
        }else{
            return getMaxReq();
        }
    }

    public static void main(String[] args) {

        short[] data = createArray(100000000);

        // Mira el número de procesadores
        System.out.println("Iniciamos calculo");
        ForkJoinPool pool = new ForkJoinPool();

        int inici=0;
        int  fi= data.length;
        MaximTask tasca = new MaximTask(data, inici, fi);

        long time = System.currentTimeMillis();
        // crea la tarea y espera que se complete
        int result1 = pool.invoke(tasca);
        // máximo
        int result=  tasca.join();
        System.out.println("Tiempo utilizado:" +(System.currentTimeMillis()-time));

        System.out.println ("El máximo es " + result);
    }


    private static short [] createArray(int size){
        short[] ret = new short[size];
        for(int i=0; i<size; i++){
            ret[i] = (short) (1000 * Math.random());
            if(i==((short)(size*0.9))){
                ret[i]=1005;
            }
        }
        return ret;
    }
}
```

Pruebe diferentes umbrales para ver que el actual es el más eficiente.

Sincronización y comunicación entre procesos

La comunicación y la sincronización son las dos acciones más importantes en la ejecución de procesos concurrentes. Cuando se están ejecutando varios procesos al mismo tiempo no podemos controlar el orden en el que terminarán ejecutándose, ya que esto dependerá de la política de planificación elegida y de los procesos que se estén ejecutando actualmente. Pero el orden de procesamiento, a veces, puede llegar a ser esencial para lograr un resultado correcto. Las técnicas que nos ayudan a controlar el orden de ejecución de los diferentes procesos se denominan técnicas de comunicación y sincronización.

Competencia y sincronización de recursos

Los procesos concurrentes se pueden clasificar como independientes o cooperantes según su grado de colaboración. Un proceso **independiente** no necesita ayuda o cooperación de otros procesos. Por otro lado, los procesos **cooperantes** están diseñados para trabajar junto con otros procesos y, por lo tanto, deben ser capaces de comunicarse e interactuar entre sí.

Estas colaboraciones implican una coordinación de todos los procesos involucrados. A veces, la coordinación requiere un cierto nivel de comunicación para estar sincronizado. Sin embargo, la comunicación detallada es bastante costosa y a menudo es suficiente para garantizar que los procesos no accedan al mismo recurso al mismo tiempo o ejecuten la misma operación. En este último caso diremos que los procesos compiten entre sí porque no hay forma de conocer el orden exacto con el que finalmente se ejecutarán los procesos.

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

Para sincronizar procesos podemos utilizar diferentes métodos:

- **Sincronismo condicional**: o condición de sincronización. Un proceso o subproceso se encuentra en un estado de ejecución y entra en un estado de bloqueo a la espera de que se cumpla una determinada condición para continuar su ejecución. Otro proceso hace que se cumpla esta condición y, por lo tanto, el primer proceso vuelve al estado de ejecución.
- **Exclusión mutua:**se produce cuando dos o más procesos o subprocesos desean acceder a un recurso compartido. Se deben establecer protocolos de ejecución para que no accedan simultáneamente al recurso.
- **Comunicación por mensajes:**en la que los procesos intercambian mensajes para comunicarse y sincronizarse. Es la comunicación típica en sistemas distribuidos y también se utiliza en sistemas no distribuidos.

Sincronización y comunicación
La colaboración entre procesos conduce a una serie de problemas clásicos de comunicación y sincronización. Por lo tanto, sincronizar y comunicar los procesos es esencial para resolverlos.

Secciones críticas
Las secciones críticas son uno de los problemas que ocurren con mayor frecuencia en la programación concurrente. Tenemos varios procesos que se ejecutan simultáneamente y cada uno de ellos tiene una parte de código que debe ejecutarse exclusivamente ya que accede a recursos compartidos como archivos, variables comunes, registros de bases de datos, etc.

La solución será forzar el acceso a los recursos a través de la ejecución de un código que llamaremos sección crítica y que nos permitirá proteger esos recursos con mecanismos que impidan la ejecución simultánea de dos o más procesos dentro de los límites de la sección crítica.

Estos algoritmos de sincronización que impiden el acceso a una región crítica por más de un hilo o proceso y que garantizan que solo un proceso estará utilizando este recurso y el resto que quieran utilizarlo estarán esperando a que se libere, se denomina **algoritmo de exclusión mutua.**
La exclusión mutua (MUTEX) es el tipo de sincronización que impide que dos procesos ejecuten simultáneamente la misma sección crítica.

Un mecanismo de sincronización en forma de código que proteja la sección crítica debe tener un formulario como el siguiente:

```
Entrada_Sección_Crítica /* Solicitud para sección crítica */

/* Código sección crítica*/

Salida_Sección_Crítica /* otro proceso puede ejecutar la sección crítica*/
```

representa la parte del código en la que los procesos solicitan permiso para entrar en la sección crítica. En su lugar, representa la parte que los procesos ejecutan cuando salen de la sección crítica liberando la sección y permitiendo que entren otros procesos. Entrada_Sección_Crítica Salida_Sección_Crítica
Para validar cualquier mecanismo de sincronización de una sección crítica, se deben 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 enclavamiento: Ningún proceso 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 del proceso.

Un error de coherencia típico cuando no hay control sobre una sección crítica se puede ilustrar con el ejemplo de dos procesos que desean modificar una variable común x. El proceso A quiere aumentarlo: x++. Proceso B disminuirlo: x–;. Si ambos procesos acuerdan leer el contenido de la variable al mismo tiempo, ambos obtendrán el mismo valor, si realizan su operación y guardan el resultado, esto será inesperado. Depende de quién guarde el valor de x último.

La tabla muestra un ejemplo similar. Un código es accesible por dos hilos o procesos, vemos que si no hay control sobre el acceso, el primer subproceso accede a las instrucciones y antes de llegar a la instrucción para aumentar la variable a++, el segundo proceso entra a ejecutar el mismo código. El resultado es que el segundo proceso toma el valor 4, por lo tanto incorrecto, ya que el primer proceso no habría aumentado la variable.

Tabla: Sección crítica

Proceso 1	**Hora**	**Proceso 2**
faneca. imprimir(a);	1	
	2	**faneca. imprimir(a);**
a=a+4;	3	
	4	**a=a+4;**

```
faneca.
print("+4=");
faneca.
println(a);
```

```
faneca.
print("+4=");
faneca.
println(a);
```

En este ejemplo asumiremos qué salida puede representar en un archivo, qué llega a la zona crítica por parámetro y que cada proceso funciona con un archivo diferente. También asumiremos que la variable a tiene un valor inicial de 4 y que es una variable compartida por ambos procesos. Puedes imaginar que las salidas serán bastante sorprendentes, ya que en ambos archivos indicaría: 4+4=12. Para evitar este problema, solo un subproceso debe ejecutar esta parte del código simultáneamente. Esta parte del código, que es susceptible a este tipo de error, debe declararse como una sección crítica para evitar este tipo de error.

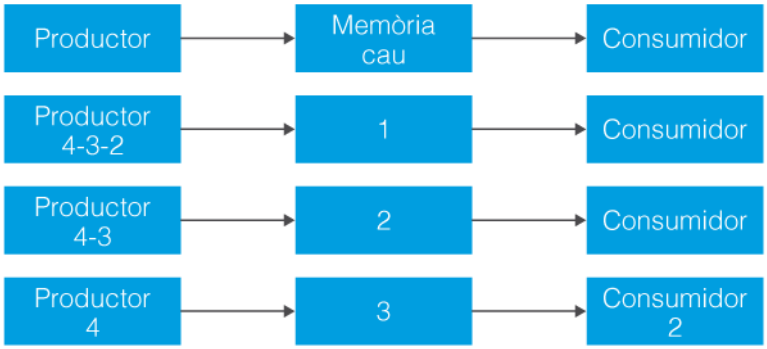
Las instrucciones que forman parte de una sección crítica deben ejecutarse como si fueran una sola instrucción. Los procesos deben sincronizarse para que un solo proceso o subproceso pueda excluir temporalmente el resto de procesos de un recurso compartido (memoria, dispositivos, etc.) de tal manera que se garantice la integridad del sistema.

Productor-consumidor

El problema productor-consumidor es un ejemplo clásico donde es necesario dar un tratamiento independiente a un conjunto de datos que se generan de forma más o menos aleatoria o al menos de una manera en la que no es posible predecir cuándo se generarán unos datos. Para evitar el uso excesivo de recursos informáticos a la espera de la llegada de los datos, el sistema prevé dos tipos de procesos: los productores, responsables de la obtención de los datos a tratar y los consumidores, especializados en el tratamiento de los datos obtenidos por los productores.

En esta situación, el productor genera una serie de datos que el consumidor recoge. Imaginemos que es el valor de una variable que el productor modifica y el consumidor la toma para su uso. El problema viene cuando el productor produce datos a un ritmo diferente al que el consumidor los recoge. El productor crea una figura y cambia la variable a su valor. Si el consumidor va más lento, el productor tiene tiempo para generar una nueva cifra y vuelve a cambiar la variable. Por lo tanto, el proceso del consumidor ha perdido el valor de los primeros datos. En el caso de que sea el consumidor el que vaya más rápido, puede ocurrir que tome el mismo dato dos veces, ya que el productor no ha tenido tiempo de sustituirlo, o que no encuentre nada que consumir. La situación puede complicarse aún más si tenemos varios procesos de producción y consumidores.

En la figura.13 podemos ver cómo dos procesos comparten un recurso común (memoria) y la ilustración del problema cuando el proceso de producción y el consumidor no están sincronizados, y el productor es más rápido que el consumidor.



FiguraProductor-consumidor

Imaginemos que en la figura.13 la memoria común es como una caja que es capaz de guardar un solo dato, un entero. Hay un proceso de producción que genera los números enteros y los deja en la caja. Mientras tanto, hay un proceso de consumo que toma el número entero de la caja.
Como es el caso, el productor deja en la memoria el número 1 y antes de que el proceso consumidor tome los datos, genera otro número, el 2, que reemplaza al anterior. El consumidor ahora toma el número 2 pero, como podemos ver, el número 1 se ha perdido, probablemente produciendo algunos resultados incorrectos.

Una forma de resolver el problema es ampliar la ubicación donde el productor escribe los datos para que sea posible mantener varios datos esperando a ser consumidos mientras los procesos del consumidor están ocupados. Es decir, los productores almacenarán los datos en una lista(array),que tradicionalmente se conoce como buffer y los consumidores los extraerán.

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

La figura.14 es el esquema de un proceso de producción que deja la información a un búfer y es tomada por un proceso de consumo.

FiguraBúfer
Lamentablemente este mecanismo no resuelve todos los problemas, ya que puede ser que un consumidor intente acceder a los datos a pesar de que el productor aún no haya escrito ninguno de ellos, puede suceder que se llene el espacio destinado a almacenar la recolección de datos porque la producción de datos siempre es mucho más rápida que los procesos del consumidor, o podría ser el caso de que dos procesos productivos coincidieran al dejar un dato o que diversos procesos de consumo intentaran acceder al tiempo.
Por lo tanto, debe haber un mecanismo que detenga el acceso a los datos de los productores y consumidores si es necesario. Una sección crítica. Desafortunadamente, no es suficiente restringir el

acceso a los datos, porque podría ser el caso de que un proceso de consumo que espera la llegada de un dato impida el acceso de los procesos de producción para que los datos nunca lleguen. Una situación como la descrita se conoce como problemasde punto muerto.
Llamamos a la situación extrema que nos encontramos cuando dos o más procesos están esperando la ejecución del otro con el fin de continuar para que nunca se desbloqueen.

También llamamos la situación que se produce cuando un proceso no puede continuar su ejecución por falta de recursos. Por ejemplo, si tuviéramos que hacer crecer el **búfer** ilimitadamente.
Para solucionar el problema, debe sincronizar el acceso al búfer. Debe accederse a la exclusión mutua para que los productores no alimenten el colchón si ya está lleno y los consumidores no puedan acceder a él si está vacío. Pero las secciones críticas del acceso de los productores en las secciones críticas del acceso de los consumidores también deberán ser independientes, evitando así el enclavamiento.

Esto obligará a la creación de un mecanismo de comunicación entre secciones críticas para que cada vez que un productor deje un dato disponible, advierta a los consumidores que pueden permanecer esperando que al menos uno de ellos comience a procesar los datos.

Lectores-escritores

Otro tipo de problema que aparece en la programación concurrente, es el que se produce cuando tenemos un recurso compartido entre varios procesos concurrentes como un archivo, una base de datos, etc. que se actualiza periódicamente. En este caso, los procesos de lectura no consumen los datos sino que solo los utilizan y por tanto se permite su consulta de forma simultánea, aunque no su modificación.

Así, los procesos que accedan al recurso compartido para leer sus contenidos se denominarán **lectores.** Por otro lado, quienes accedan a ella para modificarla recibirán el nombre de **escritores.**
Si el trabajo de lectores y escritores no se realiza de manera coordinada, podría suceder que un lector lea los mismos datos varias veces o que el escritor modifique el contenido antes de haber leído a todos los lectores, o que los datos actualizados por un escritor desperdició la actualización de otro, etc. Además, la falta de coordinación obliga a los lectores a verificar periódicamente si los escritores han realizado modificaciones, lo que aumentará el uso del procesador y, por lo tanto, reducirá la eficiencia.

Este problema de sincronización de procesos se denomina **problema lector-escritor.** Para evitar esto, es necesario asegurarse de que los procesos de los escritores accedan exclusivamente al recurso compartido y que cada modificación se notifique a los procesos lectores interesados en el cambio.

Así, los procesos de lectura pueden permanecer en suspenso hasta que se les advierta de que hay nuevos datos y puedan empezar a leer; de esta manera, los lectores están accediendo constantemente al recurso sin que el escritor haya puesto ningún dato nuevo, optimizando, de esta manera, los recursos.

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, aunque es difícil generalizar una solución ya que dependen de la complejidad, el número de secciones críticas, el número de procesos que requieren exclusión mutua y la interdependencia existente entre procesos. Aquí veremos la sincronización a través de semáforos, monitores y paso de mensajes.

Semáforos

Imagina una carretera de un solo carril que tiene que pasar por un túnel. Hay un semáforo en cada extremo del túnel que nos indica cuándo podemos pasar y cuándo no. Si el semáforo está en verde, el automóvil pasará inmediatamente y el semáforo se pondrá en rojo hasta que salga. Este símil nos introduce en la definición real de semáforo.
Los semáforos son una técnica de sincronización de memoria compartida que evita que el proceso entre en la sección crítica bloqueándola. El concepto fue introducido por el informático holandés Dijkstra para resolver el problema de la exclusión mutua y permitir resolver gran parte de los problemas de sincronización entre procesos.

- 
- Edsger Wybe Dijkstra, informático holandés. Su idea de exclusión mutua ideada durante la década de 1960 es utilizada por muchos procesadores y programadores modernos.

Los semáforos no solo controlan el acceso al tramo crítico sino que también disponen de información complementaria para decidir si bloquean o no el acceso a aquellos procesos que lo soliciten. Por ejemplo, serviría para resolver problemas simples (con poca interdependencia) de los escritores-lectores de tipo o productores-consumidores.
La solución, por ejemplo en el caso de los escritores-lectores, sería hacer que los lectores antes de consultar la sección crítica pidan permiso para acceder al semáforo, que dependiendo de si está bloqueado (rojo) o liberado (verde) detendrá la ejecución del proceso de solicitud o lo dejará continuar.
Los escritores, por su parte, antes de entrar en la sección crítica, manipularán el semáforo poniéndolo en rojo y no lo volverán a poner en verde hasta que hayan terminado de escribir y salir del apartado crítico.
De forma genérica distinguiremos 3 tipos de operaciones en un semáforo:
Los semáforos soportan 3 operaciones:

- Inicializar (inicial): esta es la operación que le permite iniciar el semáforo. La operación puede recibir un valor de parámetro que indicará si se iniciará el escape (rojo) o se liberará (verde).
- Release(s)(sendSignal):Cambia el valor interno del semáforo volvándolo verde (lo libera). Si hay procesos esperando, actívalos para que puedan terminar su ejecución.
- Block(s) (sendWait): sirve para indicar que el proceso actual quiere ejecutar la sección crítica. En caso de que el semáforo esté bloqueado, la ejecución del proceso se detendrá. También indica que es necesario poner el semáforo en rojo.
-

En el caso de que se necesite la exclusión mutua, el semáforo también contará con un sistema de espera (a través de colas de procesos) que garantice el acceso a la sección crítica de un único proceso al mismo tiempo.

En realidad, la implantación de un semáforo dependerá mucho del problema a resolver, aunque la dinámica de funcionamiento siempre es muy similar. Por ejemplo, los semáforos que apoyan el problema productor-consumidor deben implementarse utilizando la exclusión mutua tanto para liberar como para bloquear. Además, la ejecución de liberación aumentará en una unidad un medidor interno, mientras que la ejecución de bloqueo aparte de detener el proceso cuando el semáforo está bloqueado, disminuirá en una unidad el medidor interno.

Teniendo en cuenta que los procesos de producción siempre ejecutarán la operación de liberación (aumentando el medidor interno) y los procesos de consumo pedirán acceso ejecutando la operación de bloqueo (lo que disminuirá el medidor interno), es fácil ver que el valor del medidor siempre será igual a la cantidad de datos que los productores han generado sin que los consumidores aún tengan que consumir. Así podemos deducir que si en algún momento el valor alcanza el valor cero, significará que no hay datos que consumir y por tanto haremos que el semáforo esté siempre bloqueado en este caso, pero se desbloquee para que el contador aumente su valor.
Además de resolver problemas de tipo productor-consumidor o lector-escritor, podemos utilizar semáforos para gestionar problemas de sincronización donde un proceso tiene que activar la ejecución de otro o exclusión mutua asegurando que solo un proceso logrará acceder al tramo crítico porque el semáforo quedará bloqueado a la salida.

Así que si queremos sincronizar dos procesos haciendo que uno de ellos (p1) ejecute una acción siempre antes que el otro (p2), utilizando un semáforo, lo inicializaremos a 0 para asegurarnos de

que está bloqueado. La codificación del proceso p2 asegura que ante cualquier llamada a la acción que queramos controlar, solicitarás el acceso al semáforo con un . Por el contrario, en la codificación del proceso p1 siempre será necesario realizar una llamada justo después de ejecutar la acción a controlar (ver tabla.2). SendWaitsendSignal

De esta manera nos aseguraremos de que el proceso p1 siempre ejecutará la acción antes de p2. Como ejemplo, imaginamos dos procesos. Uno tiene que escribir Hola,(proceso p1) y el proceso p2 debe escribir mundo. La orden de ejecución correcta es primero p1 y luego p2, para que escribas Hello World. En caso de que el proceso P1 se ejecute antes de p2, no hay problema: Escriba Hello y haga uno en el semáforo (semáforo = 1). Cuando se ejecute el proceso p2 encontrarás semáforo=1, por lo tanto no se bloqueará, puedes hacer uno en el semáforo (semáforo=0) y escribir mundo. signalsendWait

De esta manera nos aseguraremos de que el proceso p1 siempre ejecutará la acción antes de p2. Como ejemplo, imaginamos dos procesos. Uno tiene que escribir Hola,(proceso p1) y el proceso p2 debe escribir mundo. La orden de ejecución correcta es primero p1 y luego p2, para que escribas Hello World. En caso de que el proceso P1 se ejecute antes de p2, no hay problema: Escriba Hello y haga uno en el semáforo (semáforo = 1). Cuando se ejecute el proceso p2 encontrarás semáforo=1, por lo tanto no se bloqueará, puedes hacer uno en el semáforo (semáforo=0) y escribir mundo. SignalsendWait

Tabla:

Proceso 1 (p1)	**Proceso 2 (p2)**
inicial(0)	
...	
System. out. print("Hola")	sendWait()
sendSignal()	System. out. print("mundo")

Pero, ¿qué sucede si el proceso p2 se ejecuta primero? Como encontrarás semáforos a las 0, serás bloqueado al realizar la solicitud llamando a sendWait hasta que el proceso p1 escriba Hola y haga el sendSignal,desbloqueando el semáforo. Luego se activará p2, que estaba bloqueado, volverá a poner el semáforo en rojo (semáforo = 0) y escribirá el mundo en la pantalla.
Otro ejemplo que puede ilustrar el uso de semáforos, sería el de una sucursal bancaria que gestiona nuestra cuenta corriente a la que se puede acceder desde diferentes oficinas para depositar dinero o retirar dinero. Estas dos operaciones modifican nuestro equilibrio. Serían dos funciones como las siguientes:

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

public void eliminar(float dinero) {
    float aux;
```

```
        aux=leerSaldo();
        aux=aux-dinero;
        saldo=aux;
        guardarSaldo(saldo);
    }
```

El problema viene cuando quieres obtener un ingreso simultáneamente y quieres retirar dinero. Si por un lado estamos sacando dinero de la cuenta corriente y por otro lado alguien está haciendo un depósito, se podría crear una situación anómala. Habrá dos procesos concurrentes, uno tomará dinero y el otro lo depositará. Si acceden a ambos a la vez que toman el mismo valor, imagina 100€. El proceso en el que quieres depositar dinero, quieres hacerlo con la cantidad de 300€. Y lo que quiere sacar de ella, lo quiere 30€. leerSaldo()

Si continuamos la ejecución de los dos procesos, dependiendo del orden de ejecución de las instrucciones, podemos encontrar un equilibrio diferente. Si después de leer el saldo, el proceso de pago finaliza la ejecución y ahorra el saldo, ahorraría 400 € (100 € + 300 €). Pero más tarde el proceso de retiro de dinero terminaría y ahorraría el valor de € 70 (€ 100 - € 30). Perdimos nuestros ingresos. Hay dos procesos que se están ejecutando en una sección crítica que debemos proteger en la exclusión mutua. Solo un proceso debe poder acceder a esta sección crítica y poder modificar el balance de variablescompartidas.

Para evitar el problema podemos utilizar un semáforo. Lo iniciaremos en 1, indicando el número de procesos que pueden entrar en el apartado crítico. Y tanto en el proceso de eliminación de dins como en el proceso de ingresarlo agregaremos uno al principio de las secciones críticas y otro al final. sendWait()sendSignal()

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

    public void eliminar(float dinero) {
        sendWait();
        float aux;
        aux=leerSaldo();
        aux=aux-dinero;
        saldo=aux;
        guardarSaldo(saldo);
        sendSignal();
    }
```

De esta manera, cuando un proceso entra en la sección crítica de un método, toma el semáforo. Si es 1, puede hacer el , por lo tanto, el semáforo se establecerá en 0, cerrado. Y ningún otro proceso podrá ingresar a ninguno de los métodos. Si un proceso intenta entrar, encontrarás el semáforo a las 0 y se bloqueará hasta que el proceso que tiene el semáforo haga uno, ponga el semáforo a 1 y suelte el semáforo. sendWaitsendSignal

El uso de semáforos es una forma eficiente de sincronizar procesos simultáneos. Resolver la exclusión mutua de una manera sencilla. Pero desde el punto de vista de la programación, los algoritmos son difíciles de diseñar y entender, ya que las operaciones de sincronización pueden dispersarse por el código. Por lo tanto, los errores se pueden cometer fácilmente.

Monitores

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

Las operaciones de monitor se encapsulan dentro de un módulo para protegerlas del programador. Solo se puede ejecutar un proceso dentro de este módulo.

El grado de seguridad es alto ya que los procesos no saben cómo se implementan estos módulos. El programador no sabe cómo y en qué punto se llaman las operaciones del módulo, por lo que es más robusto. Un monitor, una vez implementado, si funciona correctamente siempre funcionará bien.
Un monitor se puede ver como una habitación, cerrada con una puerta, que tiene recursos en su interior. Los procesos que quieran utilizar estos recursos deben entrar en la sala, pero con las condiciones establecidas por el monitor y solo un proceso a la vez. Cualquiera que quiera hacer uso de los recursos tendrá que esperar a que salga lo que hay dentro.

Para la encapsulación, la única acción que debe realizar el programador del proceso que quiera acceder al recurso protegido es informar al monitor. La exclusión mutua está implícita en ella. Los semáforos, por otro lado, deben implementarse con una secuencia de señal correcta y esperar para no bloquear el sistema.

Un **monitor** es un algoritmo que abstrae abstractamente datos que nos permite representar abstractamente un recurso compartido a través de un conjunto de variables que definen su estado. El acceso a estas variables solo es posible a partir de métodos de monitor.
Los monitores deben ser capaces de incorporar un mecanismo de sincronización. Por lo tanto, deben implementarse. Se pueden utilizar señales. Estas señales se utilizan para prevenir choques. Si el proceso que está en el monitor tiene que esperar una señal, se pone en un estado de espera o se bloquea fuera del monitor, lo que permite que otro proceso use el monitor. Los procesos fuera del monitor están esperando que una condición o señal vuelva a entrar.

Estas variables que son utilizadas por las señales y son utilizadas por el monitor para la sincronización se denominan **variables de condición.** Estos pueden ser manipulados con y (como semáforos). SendSignalsendWait

- sendWait: Un proceso que está esperando un evento indicado por una variable de condición abandona temporalmente el monitor y se pone en cola correspondiente a su variable de condición.
- sendSignal: Desbloquea un proceso de la cola de procesos bloqueados con la variable de condición dada y se coloca en un estado listo para ingresar al monitor. El proceso que entra no tiene por qué ser el que más tiempo ha estado esperando, sino que debe asegurarse de que el tiempo de espera de

un proceso sea limitado. Si no existe ningún proceso en cola, la operación sendSignal no tiene ningún efecto y se introducirá el primer proceso que solicite el uso del monitor.
Un monitor consta de 4 elementos:

- Variables o métodos permanentes o privados: son las variables y métodos internos del monitor a los que solo se puede acceder desde el interior del monitor. No se modifican entre dos llamadas consecutivas al monitor.
- Código de inicio: inicializa variables permanentes, se ejecuta cuando se crea el monitor.
- Métodos externos o exportados: son métodos a los que se puede acceder desde fuera del monitor por los procesos que se quieren introducir para utilizar.
- Cola de procesos: Es la cola de procesos bloqueados que espera que la señal que los libera vuelva a entrar en el monitor.

En el campo de la programación, un monitor es un objeto en el que todos sus métodos se implementan bajo exclusión mutua. En el lenguaje Java son objetos de una clase en la que se sincronizan todos sus métodos públicos.
wait, notify y notifyAll, son operaciones que permiten sincronizar el monitor.
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. Son la encapsulación de un objeto, por lo que hace que un objeto sea más seguro, más robusto y escalable.

Sincronizar en Java

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

Todos los métodos con el conmutador sincronizado se ejecutarán en exclusión mutua. El modificador garantiza que si un método sincronizado se está ejecutando, se pueda ejecutar cualquier otro método sincronizado. Pero los métodos no sincronizados pueden estar ejecutándose y con más de un proceso a la vez. Además, solo el método sincronizado puede ser ejecutado por un proceso, el resto de los procesos que quieran ejecutar el método tendrán que esperar a que finalice el proceso que lo está ejecutando.

Sincronizar métodos en Java

```
public synchronized  void metodoSincronizar {
   //código a sincronizar
}
```

Si no nos interesa sincronizar todo el método, sino solo una parte del código, utilizaremos la palabra reservada sincronizada sobre el mismo objeto que ha llamado al método en el que se encuentra la parte del código a sincronizar.

Sincronizar bloques de código en Java

```
public  void metodoNoSincronizar{
  //parte del codigo sin sincronizar
  synchronized(this){
    //parte de codigo a sincronizar
  }
```

```
  //parte de código sin sincronizar
}
```

this es el objeto que ha llamado al método, pero también podemos usar otro objeto, si queremos realizar más de una operación atómica sobre un objeto. synchroniced(objecte)
Java, por tanto, nos proporciona recursos sincronizados que nos darían la implementación de un semáforo o un monitor. Sus librerías nos proporcionan acceso en exclusión mutua y controlan los errores de bloqueo o enclavamiento que puedan ocasionarse entre 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. Es responsable de decidir qué proceso hace uso del procesador. Luego, las instrucciones u operaciones que componen el proceso se ejecutan en el procesador siguiendo un orden específico.
Una operación o grupo de operaciones debe ejecutarse como si fueran una sola instrucción. Además, no se pueden ejecutar simultáneamente con ninguna otra operación que implique un dato o recurso que utilice la primera operación.

La atomicidad de una operación es poder garantizar que dos operaciones no se puedan ejecutar simultáneamente si utilizan un recurso compartido hasta que una de ellas deje libre este recurso. Una operación atómica sólo 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 se ejecutan en el código máquina. Por otro lado, el**modelo de atomicidad** es un grupo de instrucciones, en código máquina, que se ejecuta atómicamente.

Clases de operaciones atómicas

La instrucción o tiene la siguiente secuencia de instrucciones atómicas reales: x++;x= x + 1;
1. Valor de lectura x
2. Añadir a x1
3. Guardar valor x

La atomicidad del modelo entiende la instrucción completa como atómica. x = x + 1
Dos procesos ejecutan en paralelo el siguiente ejemplo:

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

Si tenemos en cuenta que los dos procesos ejecutan en paralelo la instrucción en la atomicidad real un orden de ejecución sería el siguiente (tabla.3): x=x+1;
Tabla:
1 int x=0;

	Proceso 1 (x=x+1)	Proceso 2(x=x+1)
2	Valor de lectura x	
3		Valor de lectura x
4	Añadir a x 1	
5		Añadir a x 1
6	Valor de guardado x	
7		Valor de guardado x
8	Sistema. Fuera. printl(x);	

Dependiendo del orden de ejecución de las operaciones atómicas de un proceso y del otro el resultado o valor de x puede variar.

Clases atómicas de Java

Las clases atómicas de Java son una primera solución a los problemas de procesos concurrentes. Permiten que algunas operaciones se ejecuten como si fueran atómicas reales.
Una operación atómica tiene solo dos estados: el inicial y el resultado, es decir, se completa sin interrupciones de principio a fin.

Volátil 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 se almacenará en la memoria y no en un registro del procesador. Por lo tanto, es accesible por otro proceso o hilo para modificarlo. Por ejemplo: volatile int c;

En Java, el acceso a variables primitivas (excepto doble y larga)es atómico. Ya que también es acceso atómico a todas las variables primitivas en las que aplicamos el modificador volátil.
Java **asegura** la atomicidad del acceso a variables volátiles o primitivas, pero no a sus operaciones. Esto significa que la operación en la que es un , no es una operación atómica y por lo tanto no es un hilo seguro. x++;xint

Java incorpora, a partir de la versión 1.5, el paquete java. util.concurrente. atómicas, las clases que nos permiten convertir algunas operaciones en atómicas (hilos seguros), como aumentar, reducir, actualizar y añadir un valor. Todas las clases tienen métodos get y set que también son atómicos. Las clases que incorpora son: , , , , y actualizadores que se derivan básicamente de una variable volátil y son , y .

AtomicBooleanAtomicIntegerAtomicIntegerArrayAtomicLongAtomicLongArrayAtomicIntegerFieldUpdaterAtomicLongFieldUpdaterAtomicReferenceFieldUpdater
Por lo tanto, podemos convertir la operación o en una operación atómica. x++x=x+1
Clase counter sin operaciones atómicas

```
public class Contador {
   private int x = 0;
```

```
  public void aumenta() {
     x++;
  }
  public void disminuye() {
     x--;
  }
  public int getValorContador() {
     return x;
  }
}
```

Para convertir el ejemplo anterior a operaciones atómicas, utilizaremos los métodos de la clase de paquete java. util.concurrente. atómico. AtomicInteger. AtomicInteger

Clase de narración de historias usando clases atómicas

```
import java.util.concurrent.atomic.AtomicInteger;
  public class ContadorAtomic {
    private AtomicInteger x =new AtomicInteger(0);
    public void aumenta() {
     x.incrementAndGet();
    }
    public void disminuye() {
     x.decrementAndGet();
    }
    public int getValorContadorAtomic() {
     return x.get();
    }
}
```

En la documentación oficial puede encontrar más información sobre este paquete.
http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/atomic/package-summary.html

Colecciones simultáneas

Las colecciones son objetos que contienen varios datos del mismo tipo. Las colecciones también pueden estar involucradas en la ejecución de procesos concurrentes y ser utilizadas por diferentes procesos para que los datos contenidos en la recopilación también se compartan.
En Java para trabajar con colecciones debemos utilizar las Colecciones Framework en las que 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> serían las colecciones más importantes. Representan listas, conjuntos y colas. Map<E> también es una colección clave/valor. Cada uno con sus características de clasificación, métodos de inserción, extracción y consulta, permitan o no elementos repetidos, etc.

El problema con estas colecciones es que en la programación concurrente en la que son compartidas por diferentes procesos o subprocesos simultáneamente, podrían producir resultados inesperados. Algunos lenguajes de programación resolvieron este problema sincronizando clases. En Java, se utilizan colecciones, por ejemplo. synchronizedList(List<T> list) sobre la interfaz List<E> para sincronizar la clase que la implementa, la clase. Otras clases que implementan la interfaz List es . Java ya sincroniza esta clase. ArrayListVector

Java apareció en la versión 1.5. utilidad concurrente que incorpora clases que nos permiten resolver problemas de concurrencia de forma sencilla.

La clase cada vez que se modifica lamatriz crea en la memoria una matriz con el contenido sin procesos de sincronización. Esto hace que las consultas no sean tan caras. CopyOnWriteArrayList ArrayList es una colección de datos y si queremos que sea utilizado 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 esto es muy consultado y desactualizado, la sincronización penaliza mucho el acceso. Java crea la clase, que es la ejecución segura de subprocesos de un archivo . ArrayList**CopyOnWriteArrayList**ArrayList

La cola FIFO es una estructura de datos implementada para que los primeros elementos que entren en la estructura sean los primeros en salir de ella.

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

La clase es una cola de bloqueo, se implementa en BlockingQueue. Lo que hace es que para cada operación de inserción hay que esperar una operación de eliminación y para cada inserción hay que esperar una eliminación. **SynchronousQueue**
ConcurrentMap es otra clase que define una tabla hash con operaciones atómicas.
Puede 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 especialmente para sistemas distribuidos, son los mensajes. Recuerde que un sistema distribuido no comparte memoria, por lo tanto, no es posible comunicarse entre procesos con variables compartidas.
Este sistema de sincronización y comunicación está incluido en la mayoría de los sistemas operativos y se utiliza tanto para la comunicación entre diferentes ordenadores, como para la comunicación entre procesos de un mismo ordenador. La comunicación a través de mensajes siempre requiere un proceso de emisión y un proceso de receptor para intercambiar información. Por lo tanto, las operaciones básicas serán enviar (mensaje) y recibir (mensaje).

Dado que en esta comunicación pueden estar implicados varios procesos, es necesario permitir que las operaciones básicas identifiquen quién es el destinatario e incluso sería recomendable a veces identificar al emisor. Las operaciones básicas deben ampliarse incorporando quién está dirigiendo el mensaje o cómo el receptor podría indicar quién está esperando el mensaje.

Dependiendo de la forma de referencia del destinatario y del emisor, hablaremos de envío de mensajes en **comunicación directa** o **comunicación indirecta.**

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

send(A,message)→Enviar un mensaje para procesar A
receive(B,misstage)→Resaltrar un mensaje del proceso B

Este sistema de comunicación ofrece una gran seguridad ya que identifica los procesos que actúan en la comunicación y no introduce ningún retraso en la identificación de procesos. Por otro lado, esto también es una desventaja, ya 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 procesos. En otras palabras, el sistema operativo y el lenguaje de programación deberán admitir la gestión de procesos. También es necesario que los procesos se ejecuten en el mismo ordenador, ya que de lo contrario es imposible acceder a ellos directamente.

Cuando los procesos se ejecutan en diferentes ordenadores, será necesario interponer entre sí un sistema de mensajería que envíe y recoja mensajes independientemente del proceso emisor y receptor. Solo será necesario que ambos procesos tengan acceso al mismo sistema de intercambio de mensajes y que permita identificar de alguna manera al emisor y al receptor. Tienes que darte cuenta de que esta es la identificación interna del sistema de mensajería. En ningún caso son referencias directas a los procesos.

Las referencias a los procesos de emisión y recepción del mensaje son desconocidas en la **comunicación indirecta.** 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 consultar mensajes desde este identificador para destriñirlos. En cierto modo, es como la comunicación vehicular a través de un buzón común (el identificador). El remitente envía el mensaje a un buzón específico y el receptor recopila el mensaje del mismo buzón.

Los comandos serían los siguientes:

send(Mailbox,message)→ El proceso de remitente deja el mensaje en el buzón A
receive(Mailbox,message)→ El proceso de recepción recopila el mensaje del buzón A

Este sistema es más flexible que la comunicación directa ya que nos permite tener más de un enlace de comunicaciones entre los dos procesos y podemos crear enlaces de tipo uno a muchos, muchos a uno y muchos a muchos. En los vínculos de comunicación, uno a muchos, como los buzones de correo de los sistemas cliente/servidor, se denominan **puertos.** Estos múltiples enlaces no se podían hacer con el sistema de comunicación directa. El sistema operativo se encarga de llevar a cabo la asociación de buzones a procesos y puede hacerlo de forma dinámica o estática.
Por otro lado, si tenemos en cuenta la inmediatez en la recepción de mensajes entre procesos hablaremos de: **comunicación síncrona** o **asíncrona.**

En **comunicación síncrona,** el proceso que envía un mensaje a otro proceso está esperando recibir una respuesta. Por lo tanto, el emisor está bloqueado hasta que reciba esta respuesta. Y viceversa. Sin embargo, en **la comunicación asincrónica** el proceso que envía el mensaje continúa su ejecución sin preocuparse de si el mensaje ha sido recibido o no. Por lo tanto, debe haber un búfer aquí o los mensajes se acumulan. El problema vendrá cuando el búfer esté lleno. El proceso que envía el mensaje se bloquea hasta que encuentra el espacio dentro del búfer.
Un símil a estos dos tipos de sincronización podría ser el envío de mensajes por correo electrónico o chat. La primera, que deja los mensajes en un buzón sin preocuparse de si el receptor lo recibe o no, sería la comunicación asincrónica. Por otro lado, en un chat la comunicación es totalmente sincrónica ya que el emisor y el receptor deben coincidir en el tiempo.

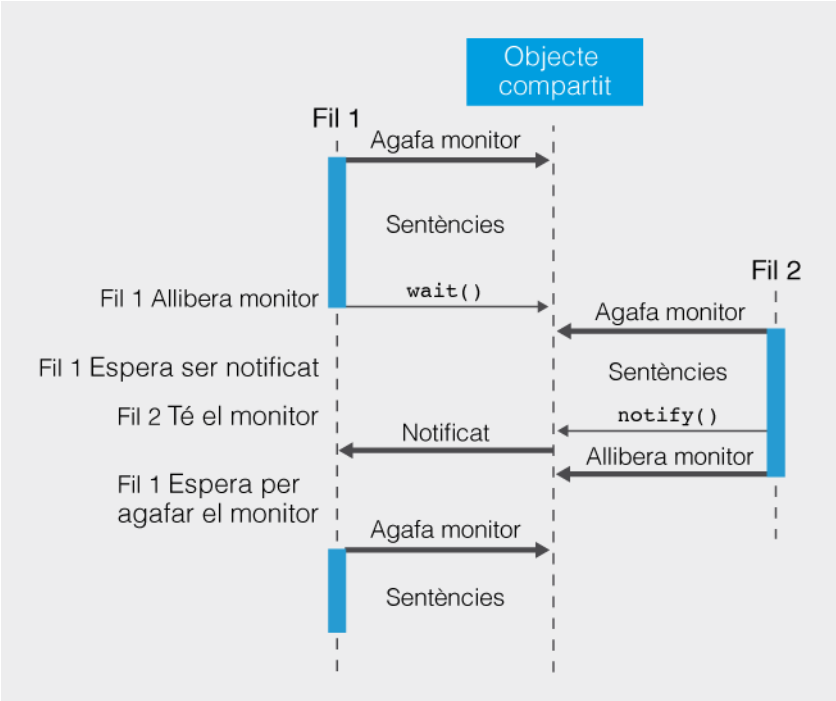
Soporte con Java

La comunicación y sincronización entre procesos de diferentes ordenadores conectados en red (sistemas distribuidos) requieren de un mecanismo de transferencia que asume la comunicación indirecta, zócalos o sockets. Estas bibliotecas cuentan con todos los mecanismos necesarios para enviar y recibir mensajes a través de la red.

Dentro de un mismo ordenador, Java cuenta con métodos como , y , que modifican el estado de un subproceso deteniendo o activando la ejecución de los procesos referenciados. Estos métodos siempre deben invocarse dentro de un bloque sincronizado (con el conmutador).
wait()notify()notifyAll()synchronize

wait(): Pone el subproceso en ejecución en modo de espera. Este subproceso sale de la zona de exclusión mutua y espera, en una cola de espera, a ser reactivado por un método o .
notify()notifyAll()
notify(): un subproceso de la cola de espera pasa al estado preparado.
notifyAll(): todos los subprocesos de la cola de espera van al estado preparado.

Cuando un proceso entra en la sección crítica, es decir, controla el monitor, ningún otro subproceso puede entrar en la sección crítica del mismo objeto que es controlado por el propio monitor. Para coordinar y comunicar, se utilizarán los métodos anteriores. Cuando no se cumplen las condiciones para continuar ejecutando un subproceso que se encuentra en la sección crítica, puede salir de la sección y permitir que otro subproceso que está en espera agarre el monitor y entre en la sección crítica. La siguiente figura(figura.15) muestra cómo dos subprocesos desean acceder a un recurso común y comunicarse para sincronizarse. Imagina que el share es una tarjeta SIM en un teléfono móvil. El hilo 1 quiere acceder a la tarjeta para hacer una llamada, tomar el monitor de compartir para tener acceso exclusivo a la tarjeta, pero la SIM aún no está activada. No se cumple la condición para poder correr. Por lo tanto, ejecuta el método que lo mantiene en espera hasta que se cumpla la condición. Cualquier otro proceso que quiera hacer la llamada y recoger el monitor hará lo mismo, ejecutará el método y permanecerá en la cola de espera. wait()wait()



Figurawait() y notify()

El hilo 2 es un proceso que se encarga de activar la tarjeta SIM. Cuando toma el monitor, tiene acceso de exclusión mutua a la tarjeta. Hace declaraciones para activar la SIM y ejecuta . Esta instrucción notifica al primer subproceso, que está en la cola de procesos en espera, que las condiciones ya se cumplen para continuar la ejecución de los procesos de llamada y liberar el monitor. La diferencia entre y es que la notificación se realiza a un hilo en espera o a todos. Finalmente, el subproceso 1 toma el monitor y ejecuta sus instrucciones de llamada. notify()notify()notifyAll()

En Java, **utilizando semáforos sincronizados,** se crean. Si añadimos los métodos, o crearemos monitores de forma fácil y eficiente. wait()notify()notifyAll()