SINCRONIZACION

INDICE

1.- Sincronización entre procesos.

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

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

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

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

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

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

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

En el caso de escritura, o lectura-escritura. El SO nos permitirá pedir un tipo de acceso de forma exclusiva al fichero. Esto significará que el proceso deberá esperar a que otros procesos lectores terminen sus accesos. Y otros procesos (lectores o escritores), esperarán a que ese proceso escritor haya finalizado su escritura.

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

1.1.- Regiones críticas.

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

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

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

Las instrucciones que forman una sección crítica, serán las mínimas. Incluirán sólo las instrucciones imprescindibles que deban ser ejecutadas de forma atómica.

Se pueden definir tantas secciones críticas como sean necesarias.

Un único proceso entra en su sección crítica. El resto de procesos esperan a que éste salga de su sección crítica. El resto de procesos esperan, porque encontrarán el recurso bloqueado. El proceso que está en su sección crítica, es el que ha bloqueado el recurso.

Al final de cada sección crítica, el recurso debe ser liberado para que puedan utilizarlo otros procesos.

Algunos lenguajes de programación permiten definir bloques de código como secciones críticas. Estos lenguajes, cuentan con palabras reservadas específicas para la definición de estas regiones. En Java, veremos cómo definir este tipo de regiones a nivel de hilo en posteriores unidades.

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

1.1.1.- Categoría de proceso cliente-suministrador.

En este caso, vamos a hacer una introducción a los procesos que podremos clasificar dentro de la categoría cliente-suministrador. Cliente. Es un proceso que requiere o solicita información o servicios que proporciona otro proceso.

Suministrador. Probablemente, te suene más el término servidor; pero, no queremos confundirnos con el concepto de servidor en el que profundizaremos en próximas unidades.

Suministrador, hace referencia a un concepto de proceso más amplio; un suministrador, suministra información o servicios; ya sea a través memoria compartida, un fichero, red, o cualquier otro recurso.

Información o servicio es perecedero. La información desaparece cuando es consumida por el cliente; y, el servicio es prestado en el momento en el que cliente y suministrador están sincronizados.

Entre un cliente y un suministrador (ojo, empecemos con un proceso de cada), se establece sincronismo entre ellos, por medio de intercambio de mensajes o a través de un recurso compartido. Entre un cliente y un servidor, la comunicación se establece de acuerdo a un conjunto mensajes a intercambiar con sus correspondientes reglas de uso; llamado protocolo.

Cliente y suministrador, son, los procesos que vimos en nuestros ejemplos de uso básico de sockets y comunicación a través de tuberías y, por supuesto, se puede extender a los casos en los que tengamos un proceso que lee y otro que escribe en un recurso compartido.

Entre procesos cliente y suministrador debemos disponer de mecanismos de sincronización que permitan que:

Un cliente no debe poder leer un dato hasta que no haya sido completamente suministrado. Así nos aseguraremos de que el dato leído es correcto y consistente.

Un suministrador irá produciendo su información, que en cada instante, no podrá superar un volumen de tamaño máximo establecido; por lo que el suministrador, no debe poder escribir un dato si se ha alcanzado ese máximo. Esto es así, para no desbordar al cliente.

Lo más sencillo, es pensar que el suministrador sólo produce un dato que el cliente tiene que consumir. ¿Qué sincronismo hace falta en esta situación?
1. El cliente tiene que esperar a que el suministrador haya generado el dato.
2. El suministrador genera el dato y de alguna forma avisa al cliente de que puede consumirlo.

Podemos pensar en dar una solución a esta situación con programación secuencial. Incluyendo un bucle en el cliente en el que esté testeando el valor de una variable que indica que el dato ha sido producido.

Como podemos ver en este gráfico el pseudocódigo del cliente incluye el bucle del que habíamos mencionado. Ese bucle hace que esta solución sea poco eficiente, ya que el proceso cliente estaría consumiendo tiempo de CPU sin realizar una tarea productiva; lo que conocemos como espera activa. Además, si el proceso suministrador quedara bloqueado por alguna razón, ello también bloquearía al proceso cliente.

En los próximos apartados, vamos a centrarnos en los mecanismos de programación concurrente que nos permiten resolver estos problemas de sincronización entre procesos de forma eficiente, llamados primitivas de programación concurrente: semáforos y monitores; y son estas primitivas las que utilizaremos para proteger las secciones críticas de nuestros procesos.

1.2.- Semáforos.

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

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

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

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

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

Semáforos binarios. Aquellos que pueden tomar sólo valores 0 ó 1. Como nuestras luces verde y roja.

Semáforos generales. Pueden tomar cualquier valor Natural (entero no negativo).

En cualquier caso, los valores que toma un semáforo representan:

Valor igual a 0. Indica que el semáforo está cerrado.

Valor mayor de 0. El semáforo está abierto.

Cualquier semáforo permite dos operaciones seguras (la implementación del semáforo garantiza que la operación de chequeo del valor del semáforo, y posterior actualización según proceda, es siempre segura respecto a otros accesos concurrentes): objSemaforo.wait(): Si el semáforo no es nulo (está abierto) decrementa en uno el valor del semáforo. Si el valor del semáforo es nulo (está cerrado), el proceso que lo ejecuta se suspende y se encola en la lista de procesos en espera del semáforo.

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

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

Para utilizar semáforos, seguiremos los siguientes pasos:

1. Un proceso padre creará e inicializará el semáforo.
2. El proceso padre creará el resto de procesos hijo pasándoles el semáforo que ha creado. Esos procesos hijos acceden al mismo recurso compartido.
3. Cada proceso hijo, hará uso de las operaciones seguras wait y signal respetando este esquema:
    1. objSemaforo.wait(); Para consultar si puede acceder a la sección crítica.
    2. Sección crítica; Instrucciones que acceden al recurso protegido por el semáforo objSemaforo.
    3. objSemaforo.signal(); Indicar que abandona su sección y otro proceso podrá entrar.

El proceso padre habrá creado tantos semáforos como tipos secciones críticas distintas se puedan distinguir en el funcionamiento de los procesos hijos (puede ocurrir, uno por cada recurso compartido).

La ventaja de utilizar semáforos es que son fáciles de comprender, proporcionan una gran capacidad funcional (podemos utilizarlos para resolver cualquier problema de concurrencia). Pero, su nivel bajo de abstracción, los hace peligrosos de manejar y, a menudo, son la causa de muchos errores, como es el interbloqueo. Un simple olvido o cambio de orden conduce a bloqueos; y requieren que la gestión de un semáforo se distribuya por todo el código lo que hace la depuración de los errores en su gestión es muy difícil.

En java, encontramos la clase Semaphore dentro del paquete java.util.concurrent; y su uso real se aplica a los hilos de un mismo proceso, para arbitrar el acceso de esos hilos de forma concurrente a una misma región de la memoria del proceso. Por ello, veremos ejemplos de su uso en siguiente unidades de este módulo.

1.3.- Monitores.

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

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

Los monitores encierran en su interior los recursos o variables compartidas como componentes privadas y garantizan el acceso a ellas en exclusión mutua.

La declaración de un monitor incluye:

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

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

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

El monitor garantiza el acceso al código interno en régimen de exclusión mutua.

Tiene asociada una lista en la que se incluyen los procesos que al tratar de acceder al monitor son suspendidos.

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

Pensemos un poco, ¿hemos utilizado objetos que pueden encajar con la declaración de un monitor aunque su tipo de dato no fuera monitor?, ¿no?, ¿seguro?

Cuando realizamos una lectura o escritura en fichero, nuestro proceso queda bloqueado hasta que el sistema ha realizado completamente la operación. Nosotros inicializamos el uso del fichero indicando su ruta al crear el objeto, por ejemplo, FileReader; y utilizamos los métodos expuestos por ese objeto para realizar las operaciones que deseamos con ese fichero. Sin embargo, el código que realmente realiza esas operaciones es el implementado en la clase FileReader. Si bien, esos objetos no proporcionan exclusión mutua en los accesos al recurso; o, por lo menos, no en todos sus métodos. Aún así, podemos decir que utilicemos objetos de tipo monitor al acceder a los recursos del sistema, aunque no tengan como nombre Monitor.

Las ventajas que proporciona el uso de monitores son:

Uniformidad: El monitor provee una única capacidad, la exclusión mutua, no existe la confusión de los semáforos.

Modularidad: El código que se ejecuta en exclusión mutua está separado, no mezclado con el resto del programa.

Simplicidad: El programador o programadora no necesita preocuparse de las herramientas para la exclusión mutua .

Eficiencia de la implementación: La implementación subyacente puede limitarse fácilmente a los semáforos.

Y, la desventaja:

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

1.3.1.- Monitores: Lecturas y escrituras bloqueantes en recursos compartidos.

Recordemos, el funcionamiento de los procesos cliente y suministrador podría ser el siguiente:

Utilizan un recurso del sistema a modo de buffer compartido en el que, el suministrador introduce elementos; y, el cliente los extrae.

Se sincronizarán utilizando una variable compartida que indica el número de elementos que contiene ese buffer compartido, cuyo tamaño máximo será N.

El proceso suministrador, siempre comprueba antes de introducir un elemento, que esa variable tenga un valor menor que N. Al introducir un elemento incrementa en uno la variable compartida.

El proceso cliente, extraerá un elemento del buffer y decrementará el valor de la variable; siempre que el valor de la variable indique que hay elementos que consumir.

Los mecanismos de sincronismo que nos permiten el anterior funcionamiento entre procesos, son las lecturas y escrituras bloqueantes en recursos compartidos del sistema

(streams).

Si nos damos cuenta, hasta ahora sólo hemos hablado de un proceso Suministrador y un proceso cliente. El caso en el que un suministrador tenga que dar servicio a más de un cliente, aprenderemos a solucionarlo utilizando hilos o threads, e implementando esquemas de cliente-servidor, en las próximas unidades. No obstante, debemos tener claro, que el sincronismo cuando hay una información que genera un proceso (o hilo), y que recolecta otro proceso (o hilo) atenderá a las características que hemos descrito en estos aparatados.

1.4.- Memoria compartida.

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

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

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

1.5.- Cola de mensajes.

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

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

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

En el paso de mensajes asíncrono, el proceso que envía, no espera a que el mensaje sea recibido, y continúa su ejecución, siendo posible que vuelva a generar un nuevo mensaje y a enviarlo antes de que se haya recibido el anterior. Por este motivo se suelen emplear buzones o colas, en los que se almacenan los mensajes a espera de que un proceso los reciba. Generalmente empleando este sistema, el proceso que envía mensajes sólo se bloquea o para, cuando finaliza su ejecución, o si el buzón está lleno.

Para conseguir esto, estableceremos una serie de reglas de comunicación (o protocolo) entre emisor y receptor, de forma que el receptor pueda indicar al emisor qué capacidad restante queda en su cola de mensajes y si está lleno o no.

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

2.- Requisitos: seguridad, vivacidad, eficiencia y reusabilidad.

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

Todo programa concurrente debe satisfacer dos tipos de propiedades:

Propiedades de **seguridad** ("safety"): estas propiedades son relativas a que en cada instante de la ejecución no debe haberse producido algo que haga entrar al programa en un estado erróneo:

Dos procesos no deben entrar simultáneamente en una sección crítica.
Se respetan las condiciones de sincronismo, como: el consumidor no debe consumir el dato antes de que el productor los haya producido; y, el productor no debe producir un dato mientras que el buffer de comunicación esté lleno.

Propiedades de **vivacidad** ("liveness"): cada sentencia que se ejecute conduce en algún modo a un avance constructivo para alcanzar el objetivo funcional del programa. Son, en general, muy dependientes de la política de planificación que se utilice. Ejemplos de propiedades de vivacidad son:

No deben producirse bloqueos activos (livelock). Conjuntos de procesos que ejecutan de forma continuada sentencias que no conducen a un progreso constructivo.

Aplazamiento indefinido (starvation): consiste en el estado al que puede llegar un programa que aunque potencialmente puede avanzar de forma constructiva. Esto puede suceder, como consecuencia de que no se le asigna tiempo de procesador en la política de planificación; o, porque en las condiciones de sincronización hemos establecido criterios de prioridad que perjudican siempre al mismo proceso.

Interbloqueo (deadlock): se produce cuando los procesos no pueden obtener, nunca, los recursos necesarios para finalizar su tarea.

Es evidentemente, que también nos preocuparemos por diseñar nuestras aplicaciones para que sean eficientes:

No utilizarán más recursos de los necesarios.

Buscaremos la rigurosidad en su implementación: toda la funcionalidad esperada de forma correcta y concreta.

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

Implementar el código de forma modular: definiendo clases, métodos, funciones, ...

Documentar correctamente el código y el proyecto.

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

2.1.- Dificultades en la depuración.

Cuando estamos programando aplicaciones que incluyen mecanismos de sincronización y acceden a recursos de forma concurrente junto con otras aplicaciones. A la hora de depurarlas, nos enfrentaremos a:

Los mismos problemas de depuración de una aplicación secuencial.

Además de nuevos errores de temporización y sincronización propios de la programación concurrente.

Los programas secuenciales presentan una línea simple de control de flujo. Las operaciones de este tipo de programas están estrictamente ordenados como una secuencia temporal lineal.

El comportamiento del programa es solo función de la naturaleza de las operaciones individuales que constituye el programa y del orden en que se ejecutan.

En los programas secuenciales, el tiempo que tarda cada operación en ejecutarse no tiene consecuencias sobre el resultado.

Para validar un programa secuencial se necesitaremos comprobar:

La correcta respuesta a cada sentencia.

El correcto orden de ejecución de las sentencias.

Para validar un programa concurrente se requiere comprobar los mismos aspectos que en los programas secuenciales, además de los siguientes nuevos aspectos:

Las sentencias se pueden validar individualmente solo si no están involucradas en el uso de recursos compartidos.

Cuando existen recursos compartidos, los efectos de interferencia entre las sentencias concurrentes pueden ser muy variados y la validación es muy difícil. Comprobaremos la corrección en la definición de las regiones críticas y que se cumple la exclusión mutua.

Al comprobar la correcta implementación del sincronismo entre aplicaciones; que es forzar la ejecución secuencial de tareas de distintos procesos, introduciendo sentencias explícitas de sincronización. Tendremos en cuenta que el tiempo no influye sobre el resultado.

El problema es que las herramientas de depuración no nos proporcionan toda la funcionalidad que quisiéramos para poder depurar nuestros programas concurrentes.

Una de las nuevas situaciones a las que nos enfrentamos es que a veces, los errores que parecen estar sucediendo, pueden desaparecer cuando introducimos código para tratar de identificar el problema.

Nos damos cuenta de la complejidad que entraña depurar el comportamiento de aplicaciones concurrentes, es por ello, que al diseñarlas, tendremos en cuenta los patrones de diseño, que ya están diseñados resolviendo errores comunes de la concurrencia. Podemos verlos como 'recetas', que nos permiten resolver los problemas 'tipo' que se presentan en determinadas condiciones de sincronismo y/o en los accesos concurrentes a un recurso.

3.- Programación paralela y distribuida.

Dos procesos se ejecutan de forma paralela, si las instrucciones de ambos se

están ejecutando realmente de forma simultánea. Esto sucede en la actualidad en sistemas que poseen más de un núcleo de procesamiento.

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

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

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

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

En la computación paralela y distribuida:

Cada procesador tiene asignada la tarea de resolver una porción del problema.

En programación paralela, los procesos pueden intercambiar datos, a través de direcciones de memoria compartidas o mediante una red de interconexión propia.

En programación distribuida, el intercambio de datos y la sincronización se realizará mediante intercambio de mensajes.

El sistema se presenta como una unidad ocultando la realidad de las partes que lo forman.