Tema 9

Fundamentos de los sistemas multiprocesadores

A pesar de los constantes avances tecnológicos, la tecnología del silicio parece que está llegando a su límite. Si se pretende resolver problemas cada vez más complejos y de mayores dimensiones se deben explorar nuevas alternativas tecnológicas. Una de estas es el paralelismo, que plantea obtener mayor rendimiento distribuyendo la carga computacional entre varias CPUs trabajando simultáneamente.

9.1. Introducción

Tradicionalmente, el computador se ha entendido como una máquina secuencial que requiere que, aun cuando se utilizan técnicas como segmentación, el procesador ejecute los programas procesando instrucciones máquina de una en una. Sin embargo, esto no es así. Como hemos visto en temas anteriores, un procesador superescalar posee varias unidades de ejecución y puede aprovechar el paralelismo implícito entre instrucciones para ejecutar en paralelo varias instrucciones del mismo programa.

Los **procesadores paralelos** son sistemas que interconectan varios procesadores para cooperar en la ejecución de un programa con el fin de aumentar las prestaciones y sacar más partido del paralelismo existente. Un sistema multiprocesador saca partido del denominado paralelismo de alto nivel, por contraposición a las técnicas ya estudiadas en temas anteriores, que utilizan el paralelismo a bajo nivel en sistemas de un solo procesador, tales como cauces segmentados o superescalares.

9.2. Clasificación

Las cuatro categorías definidas por Flynn (1972) se basan en el número de instrucciones concurrentes y en los flujos de datos disponibles en la arquitectura:

- Una secuencia de instrucciones y una secuencia de datos (**SISD**: *Single Instruetion, Single Data*): Un único procesador interpreta una única secuencia de instrucciones para procesar los datos almacenados en una única memoria. No explota el paralelismo a nivel de instrucción. Máquinas monoprocesador.

- Una secuencia de instrucciones y múltiples secuencias de datos (**SIMD**: *Single Instruction, Multiple Data*): Una única instrucción controla de forma simultánea y sincronizada un cierto número de elementos de proceso. Cada elemento tiene una memoria asociada, de forma que cada elemento ejecuta la misma instrucción con

diferentes datos. Los procesadores vectoriales y matriciales pertenecen a esta categoría.

- Múltiples secuencias de instrucciones y una secuencia de datos (**MISD**: *Multiple Instruction, Single Data*): Se transmite una secuencia de datos a varios procesadores, cada uno de los cuales ejecuta una instrucción diferente sobre los mismos. Esta estructura nunca se ha implementado.

- Multiples secuencias de instrucciones y múltiples secuencias de datos (**MIMD**: *Multiple Instructions, Multiple data*): Un conjunto de procesadores ejecuta simultáneamente instrucciones diferentes sobre conjuntos de datos diferentes. Es el caso de los sistemas distribuidos, bien explotando un único espacio compartido de memoria, o bien uno distribuido.

De las cuatro categorías, las SISD y la MIMD dan lugar a implementaciones paralelas.

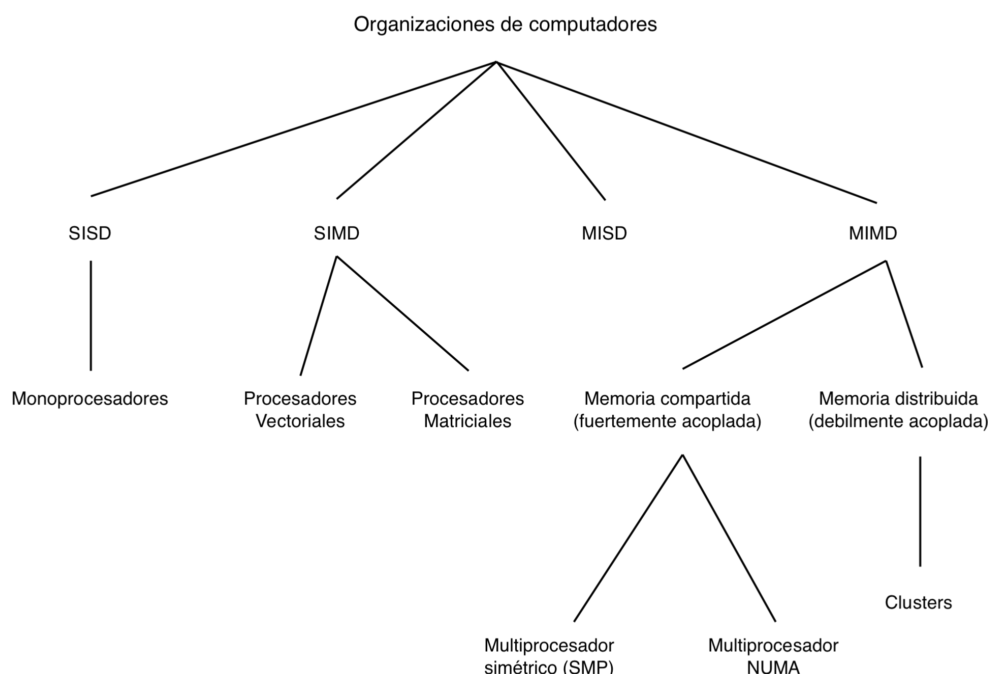
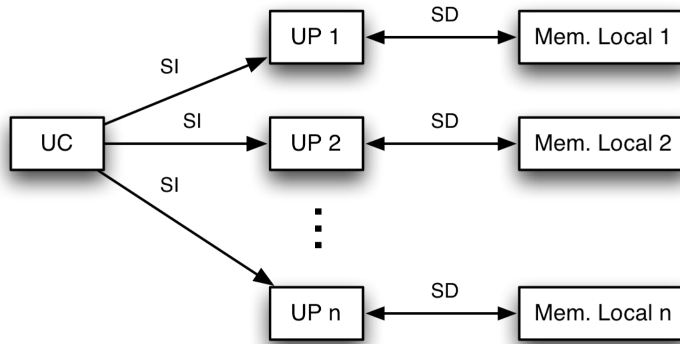


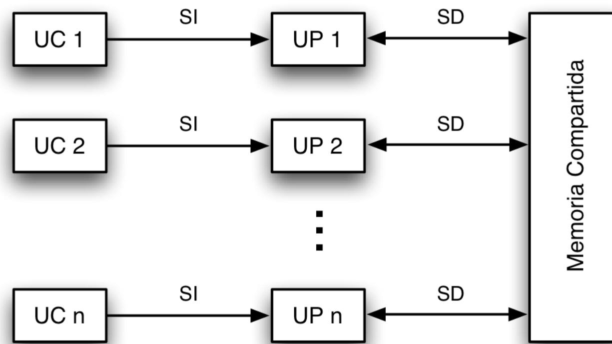
figura 9.1. Clasificación de las arquitecturas paralelas.

Los computadores MIMD se subdividen según el mecanismo de comunicación entre los procesadores (Figura 9.1). Si los procesadores comparten una **memoria común**, cada procesador accede a programas y datos a través de dicha memoria común compartida, que además se utiliza para la comunicación entre procesadores. La forma más común de este tipo de sistemas es el denominado 'multiprocesador simétrico (SMP)'. En un SMP, varios procesadores comparten una única memoria mediante un bus compartido u otro tipo de mecanismo de interconexión. Una característica distintiva de estos sistemas es que el tiempo de acceso a memoria principal es aproximadamente el mismo para cualquier procesador.

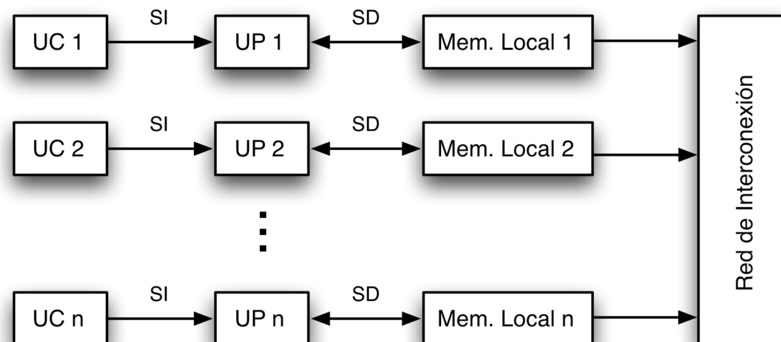
a) SISD



b) SIMD con memoria distribuida.



c) MIMD con memoria compartida.



d) MIMD con memoria distribuida.

figura 9.2. Tipos de organizaciones de computador.

Un desarrollo más reciente es la organización 'acceso no uniforme a memoria (NUMA)', en la cual el tiempo de acceso a diferentes zonas de memoria puede diferir considerablemente. Finalmente, la organización conocida como 'cluster' interconecta un conjunto de computadores independientes (monoprocesadores o SMP) mediante interconexiones fijas o algún tipo de red.

9.2.1 Organizaciones paralelas

La figura 9.2 muestra los esquemas de cada una de las clases de la clasificación de Flynn, donde UC es la unidad de control, UM la unidad de memoria, UP la unidad de proceso, SI una secuencia de instrucciones y SD una secuencia de datos.

9.3. Principales Características de los Sistemas Multiprocesadores

Una vez clasificados por el tipo de paralelismo, hay otras clasificaciones importantes que afectan a los sistemas con múltiples procesadores son las que hacen referencia a:

- Estructura lógica
- Estructura física
- Modo de interacción

9.3.1. Estructura lógica

Se entiende por **estructura lógica** el modo de distribuir la responsabilidad del control entre los diversos elementos del sistema. Es decir, define la relación entre los diversos elementos de un sistema multiprocesador.

Las dos relaciones lógicas más elementales son la **vertical** y la **horizontal**. En un sistema vertical los elementos se estructuran jerárquicamente, implicando una relación **maestro-esclavo**. En un sistema horizontal los elementos mantienen una relación de igualdad desde el punto de vista lógico, implicando una relación **maestro-maestro** (o *peer-to-peer*).

9.3.1.1. Organización vertical

Una organización vertical contiene un único maestro y uno o varios esclavos, siendo sus características principales:

1) Desde el punto de vista lógico no todos los elementos son iguales.
2) En un instante determinado sólo un elemento puede actuar como maestro, aunque otros elementos pueden tener la capacidad de llegar a actuar como maestros en otro momento.
3) Todas las comunicaciones entre los procesadores deben realizarse a través del maestro, o bien deben ser iniciadas por él.

Los sistemas verticales pueden tener más de un nivel de estructura maestro-esclavo. Esto quiere decir que uno de los esclavos puede actuar a su vez como maestro de otros elementos, dando lugar a una configuración de tipo **piramidal**.

9.3.1.2. Organización horizontal

Los sistemas así organizados requieren una coordinación más sofisticada. Sus características principales son:

1) Desde el punto de vista lógico todos los elementos son iguales.

2) Cualquier elemento puede comunicarse con cualquier otro del sistema.

En general, los sistemas horizontales son más flexibles que los verticales y son capaces de repartirse las tareas a ejecutar de una forma más dinámica.

9.3.2. Estructura física

Se denomina estructura física de un sistema multiprocesador a la forma de realizar el intercambio de información entre los procesadores que lo conforman. La estructurá física depende tanto del modo de **transferencia de los datos** como de la **interconexión topológica** entre dichos procesadores.

9.3.2.1. Transferencia de datos

Los procesadores pueden realizar transferencias de datos a través de una estructura de **memoria común** o de una **estructura de bus.** En la estructura de memoria común los procesadores no tienen acceso directo entre sí y utilizan la memoria para intercambiar los datos. En la estructura de bus, se establece un camino de comunicación entre todos los procesadores y, en general, la transferencia de datos es inicializada y ejecutada de forma distribuida.

9.3.2.2. Interconexión topológica

Físicamente, hay muchas formas de conectar N procesadores en un sistema, pero. A la hora de elegir una interconexión hay que tener en cuenta dos factores importantes: la **capacidad de expansión y la fiabilidad.** Una conexión **expandible** es aquella que *facilita la adición de más procesadores sin afectar a la estructura existente*. Una conexión **fiable** es aquella que *proporciona un camino alternativo de comunicación entre procesadores para utilizarlo en caso de fallo del camino directo*.

Las estructuras topológicas de interconexión más comunes son:

- Estructura en lazo cerrado
- Estructura completa
- Estructura bus común

- Estructura en estrella

Estructura lazo cerrado

Esta topología, como se muestra en la figura 9.3a, consiste en varios **elementos procesadores** (PE), cada uno de los cuales se conecta a dos procesadores **vecinos** a través de **caminos de comunicación.**

El **tráfico** en un lazo cerrado puede ser bidireccional. Sin embargo, en la práctica suelen utilizarse los lazos unidireccionales.

En un lazo unidireccional, un vecino de un elemento procesador es considerado como **vecino fuente,** y el otro como **vecino destino.** Un procesador recibe mensajes únicamente de su vecino fuente y los envía exclusívamente a su vecino destino. Cuando un elemento procesador envía un mensaje, dicho mensaje indica cual debe ser el procesador de destino. Los mensajes circulan alrededor del lazo, desde el procesador fuente al destino, con los procesadores intermedios actuando como **unidades buffer,** transparentes a la comunicación, ya que envían los mensajes que reciben y de los cuales no son su destinatario.

Nótese que se pueden insertar procesadores al lazo sin que apenas afecte al flujo de mensajes. Por lo tanto, el sistema es fácilmente expandible y, además, la expansión no resulta cara. Sin embargo, la fiabilidad del sistema es baja ya que un simple fallo causará la parada de la comunicación.

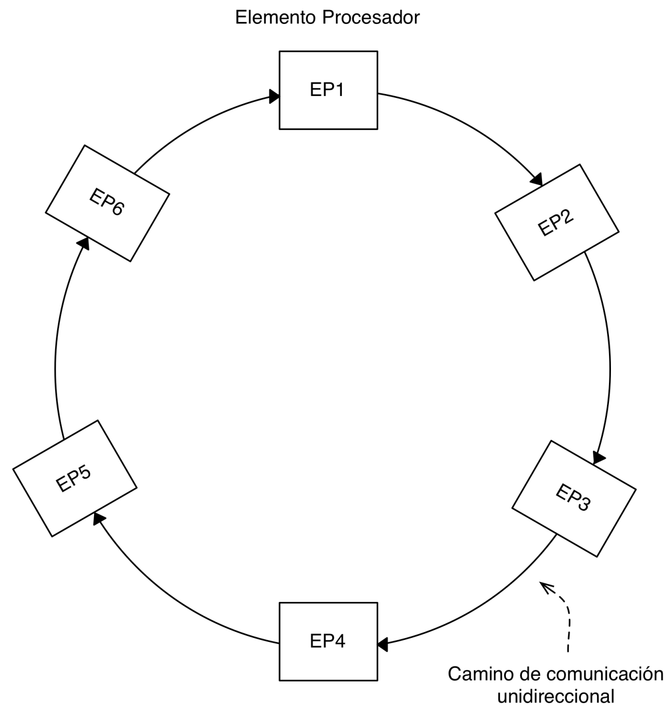
En una estructura en lazo cerrado, por cada camino de comunicación sólo puede circular un mensaje. Esto representa un problema de **cuello de botella**, por que si un procesador quiere utilizar un camino que está ocupado, tiene que esperar a que éste se desocupe.

Estructura completa

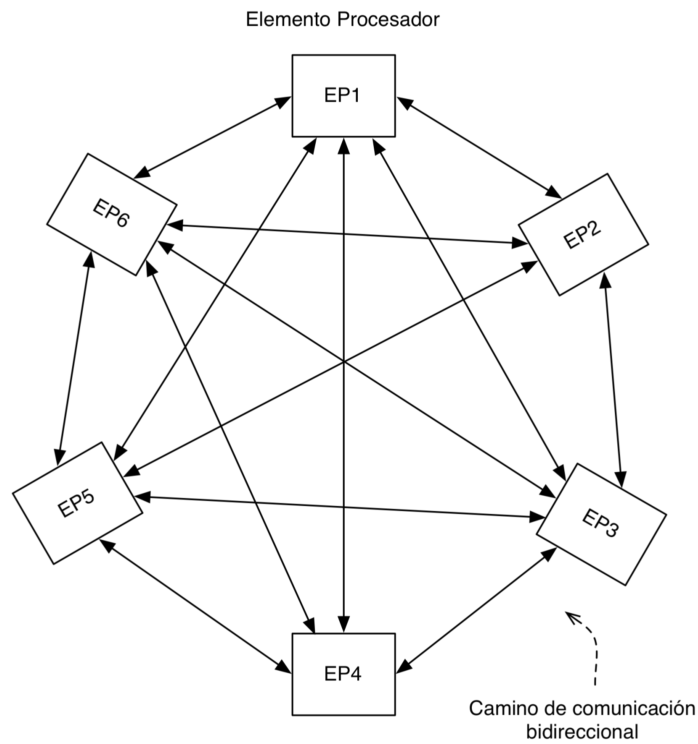
Conceptualmente, la estructura completa, mostrada en la figura 9.3b, es la más simple de diseñar: Cada procesador se conecta a todos los demás mediante un camino bidireccional.

De esta forma, los procesadores realizan la transmisión de mensajes únicamente por el camino de comunicación que lo une con el procesador destino del mensaje. Para ello, el procesador fuente debe escoger el camino correspondiente al procesador destino, dentro de todos los posibles, y todos los procesadores deben estar capacitados para recibir mensajes de los múltiples caminos conectados a ellos.

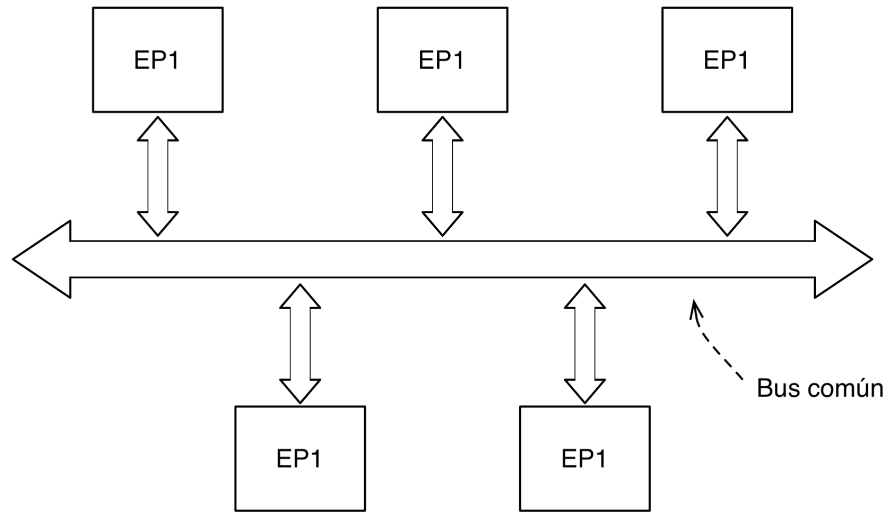
El principal inconveniente de los sistemas con estructura completa es su costosa expansibilidad. En un sistema con n procesadores, la adición al sistema del n-ésimo procesador requiere la adición de (n - 1) caminos de comunicación entre él y los demás procesadores del sistema, y éstos, a su vez, deben ser capaces de aceptar al nuevo procesador como fuente de datos. Esto supone una limitación, dado que el número de interconexiones, o **interfaces**, de los procesadores que componen el sistema acotan el tamaño del sistema multiprocesador. Es decir, si un procesador posee (M -1) puertas de comunicación, también llamadas **puertos,** el número máximo de procesadores que puede admitir el sistema es M.



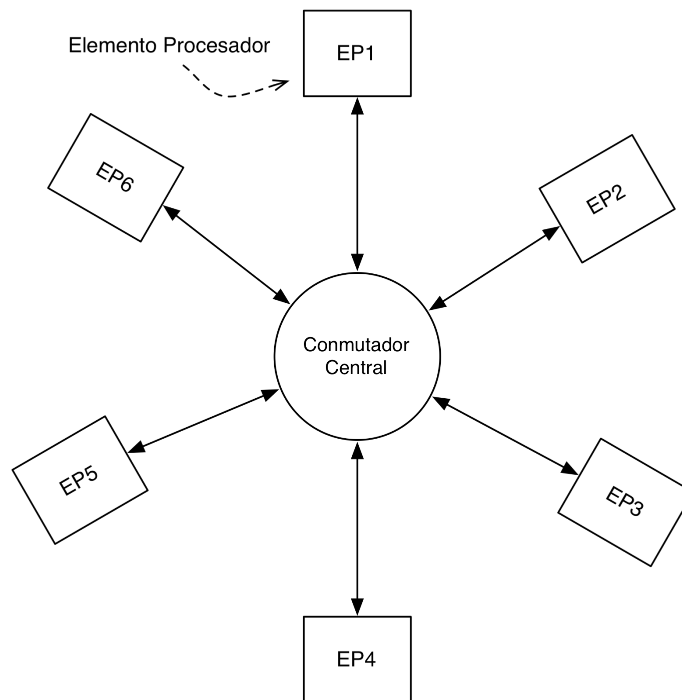
a) Estructura en lazo cerrado.



b) Estructura completa.



c) Estructura bus común



d) Estructura estrella

figura 9.3. Diversas estructuras lógicas de interconexión.

La fiabilidad del sistema es muy buena ya que, aunque se produzca un fallo en un elemento procesador o en un camino de comunicación, el resto del sistema sigue funcionando.

Por otra parte, la estructura completa no presenta nunca el problema de cuello de botella, puesto que el camino de comunicación entre dos procesadores es dedicado, no compartido.

Estructura bus común

Tal como muestra la figura 9.3c, en la estructura de bus común los diversos elementos procesadores del sistema están conectados a un mismo bus.

Como el bus no admite simultaneidad en su uso, se requiere un mecanismo adicional que administra las peticiones de uso, resolviendo en cada momento a qué procesador se le asigna el bus para realizar la siguiente transferencia: *árbitro de bus*. Esta característica hace del bus el *cuello de botella* del sistema que, en la práctica, limita el número máximo de procesadores del sistema (aunque esto está en función del número de accesos por unidad de tiempo que requiere cada procesador para realizar su tarea).

La fiabilidad del sistema depende del elemento que falle. Si falla un elemento procesador, el sistema seguirá funcionando. Si falla el bus, el sistema dejará de funcionar.

Estructura en estrella

La estructura en estrella, como se muestra en la figura 9.3d, está conformada por varios elementos procesadores que se conectan a un **conmutador central o *switch*** mediante un camino bidireccional. Para cualquier procesador el conmutador central actúa, aparentemente, como fuente y destino para todos los mensajes. Así, el número máximo de procesadores del sistema multiprocesador depende del número de puertos del conmutador central.

La fiabilidad del sistema dependerá del elemento que falla. Si falla un elemento procesador, el sistema puede seguir funcionando, pero sí falla el conmutador central, el sistema se parará.

9.3.3. Modos de interacción

Los sistemas con múltiples procesadores pueden clasificarse según el **acoplamiento,** que indica la *capacidad de compartir recursos y la naturaleza de la intercomunicación de los distintos procesadores*, de la siguiente forma:

a) Sistemas débilmente acoplados

b) Sistemas fuertemente acoplados

c) Sistemas medianamente acoplados.

9.3.3.1. Sistemas débilmente acoplados

Son las redes de computadores y se configuran con varios computadores convencionales que pueden intercomunicarse a grandes velocidades (típicamente mediante Ethernet).

El reparto de trabajo entre los computadores de la red lo realiza un único sistema operativo.

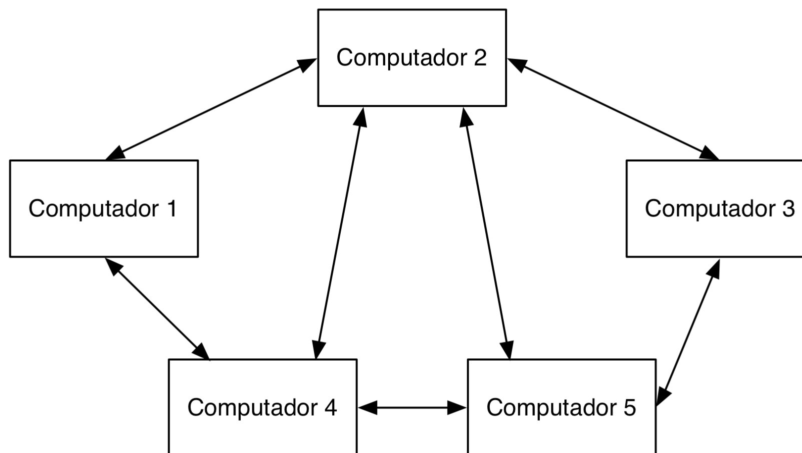


figura 9.4. Sistema débilmente acoplado.

Las características más importantes de los sistemas débilmente acoplados son:

1) **Computadores autónomos.** Los computadores son independientes y pueden estar geográficamente dispersos.
2) **Protocolo de comunicación.** La comunicación sigue un rígido protocolo.
3) **Comunicación serie.** Utilizando líneas de alta velocidad.
4) **Accesibilidad.** Desde cualquier computador se puede acceder a todos los demás.
5) **Eficacia.** El funcionamiento resulta eficiente cuando las interacciones de los procesos de los diferentes computadores es mínima.

9.3.3.2. Sistemas fuertemente acoplados

A los sistemas fuertemente acoplados se les conoce como **sistemas multiprocesadores,** y todos los procesadores que lo forman pueden utilizar todos los recursos del sistema.

Sus características más importantes son:

1) **Memoria común.** Todos los procesadores del sistema pueden acceder a una memoria principal común, aunque cada uno de ellos pueda también tener una memoria de datos propia.

2) **Entrada/Salida.** Todos los procesadores del sistema comparten el acceso a los dispositivos de entrada/salida.

3) **Sistema operativo común.** El sistema se controla mediante un sistema operativo, que regula las interacciones entre procesadores y programas.

Los sistemas fuertemente acoplados deben disponer de un mecanismo de sincronización entre procesadores. En general, todos los procesadores deben ser iguales, formando así una configuración simétrica. En la figura 9.5 se muestra la estructura básica de un sistema multiprocesador fuertemente acoplado.

En este tipo de sistemas es posible que los procesadores originen **conflictos** en el acceso a memoria principal o a los dispositivos I/O. Estos conflictos deben ser minimizados por el sistema operativo y por la estructura de interconexión.

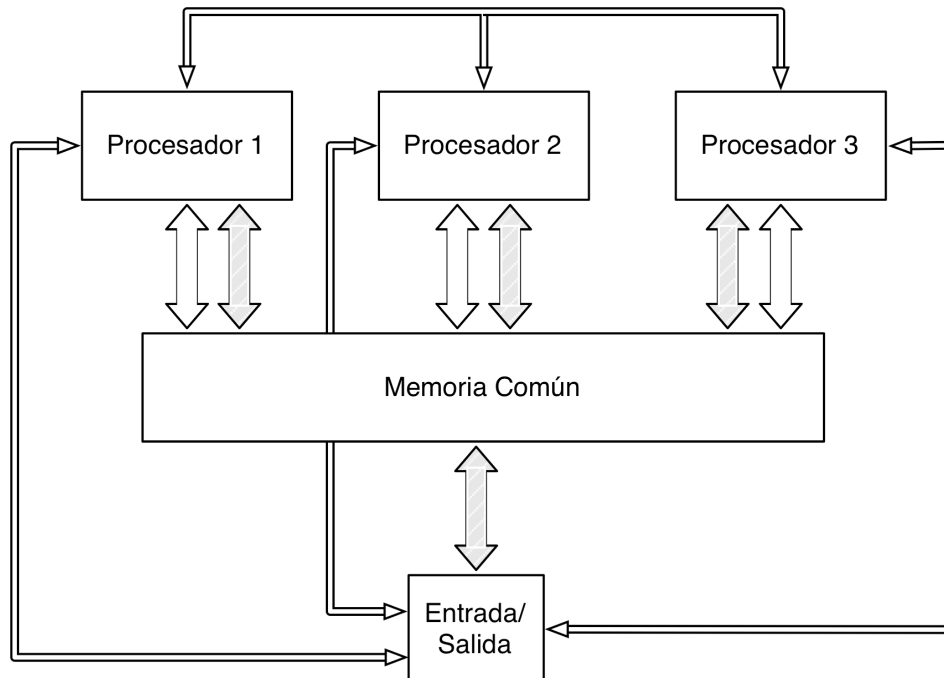


figura 9.5. Estructura básica de un multiprocesador fuertemente acoplado.

9.3.3.3. Sistemas medianamente acoplados

También denominados **sistemas distribuidos,** combinan las cualidades de los sistemas fuertemente y debilmente acoplados: Existe un mapa de memoria único y un sólo sistema óperativo, sirviendo la memoria como elemento de comunicación entre los diferentes procesadores.

9.4. Aspectos relacionados con el diseño de multiprocesadores

9.4.1. Capacidad de un sistema multiprocesador

La capacidad de un sistema multiprocesador depende de varios factores como la relación coste-rendimiento, la cantidad de procesado y la posibilidad de compartir recursos.

9.4.1.1. Relación coste-rendimiento

Al aumentar el número de procesadores de un sistema multiprocesador, aumenta el rendimiento y sus posibilidades respecto a un sistema monoprocesador, pero también debe considerarse el aumento de coste. Es deseable alcanzar un equilibrio en la relación entre coste y rendimiento.

9.4.1.2. Cantidad de procesado

La ***cantidad de procesado*** de un sistema se define como la *inversa del tiempo que el sistema requiere para ejecutar un conjunto de algoritmos dados*, y se mide en número de operaciones que se ejecutan por unidad de tiempo.

Idealmente, la cantidad de procesado aumenta con la adición de procesadores al sistema. Sin embargo, en la práctica el comportamiento es diferente, aparece un efecto de saturación y la relación entre la cantidad de procesado y número de procesadores sigue una tendencia como la mostrada en la curva real de la figura 9.6.

El **efecto de saturación** se define como la *disminución de la cantidad de procesado cuando aumenta el número de procesadores*. Puede observarse que la curva real sigue a la ideal para un bajo número de procesadores, pero cuantos más procesadores se añaden, más se aparta de la curva ideal, disminuyendo la cantidad de procesado. Este efecto se debe a que las peticiones de uso de los recursos compartidos aumentan conforme aumenta el número de procesadores del sistema, y, por tanto, las esperas para utilizar dichos recursos crecen hasta que, pasado el punto de saturación, un incremento supone un decremento del número de operaciones ejecutadas por unidad de tiempo.

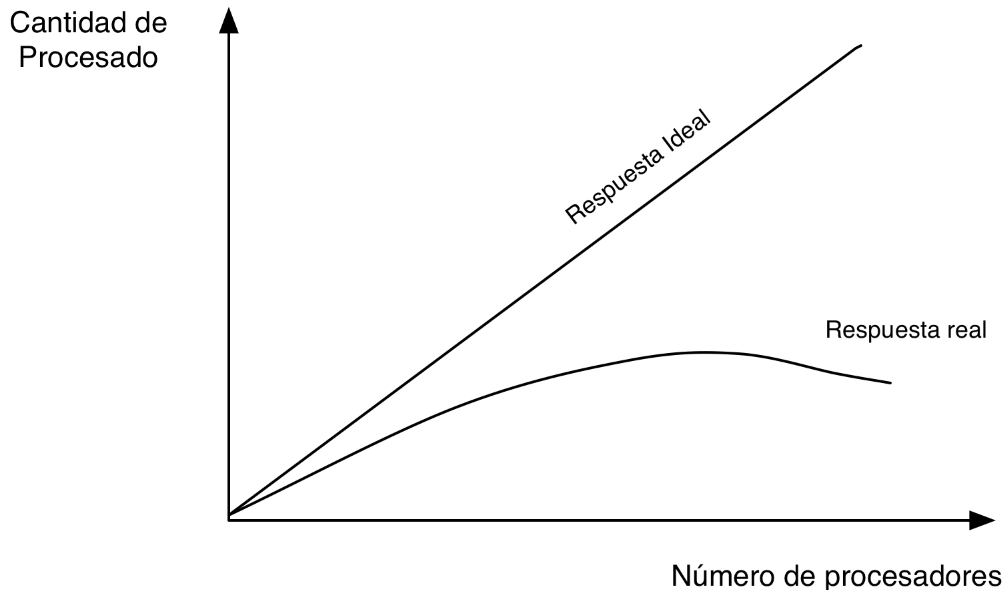


figura 9.6. Ilustración del efecto de saturación.

Para lograr trabajar en la parte más lineal de la curva, es decir, en un punto en el que la cantidad de procesado real se aproxime a la ideal, hay que reducir las peticiones de acceso a los recursos compartidos y la comunicación entre procesadores. Para ello, la mejor solución para ello es dividir el trabajo principal en pequeñas tareas independientes.

9.4.1.3. Recursos compartidos

La compartición de recursos es una de las características típicas de los sistemas multiprocesadores. Los tipos de recursos que pueden ser compartidos varían ampliamente.

9.4.2. Fiabilidad

La **fiabilidad** se define, para un sistema monoprocesador, como *la probabilidad de que un sistema funcione correctamente durante un intervalo temporal (de 0 a* Δt*)*, considerando t = 0 el instante en el que el sistema comienza a funcionar.

No obstante, en sistemas multiprocesadores la ejecución de un trabajo puede requerir la cooperación de varios procesadores. En este caso la definición clásica de fiabilidad es demasiado general, siendo más apropiado definirla como *la probabilidad de ejecutar una determinada tarea, bajo unas determinadas condiciones, en un tiempo dado*.

9.4.2.1. Sistemas con tolerancia a fallos

Para analizar la fiabilidad, la mayoría de sistemas pueden clasificarse como **redundantes** o **no redundantes**. En los sistemas no redundantes cada componente debe funcionar correctamente para que el sistema funcione. En los sistemas redundantes habrá redundancia (duplicidad) de parte o de todo el sistema. Esto asegura que el funcionamiento de las partes redundantes cuando haya un fallo.

Así, un sistema tolerante a fallos es aquel que es capaz de detectar y superar un mal funcionamiento del equipo físico y/o los errores del sistema lógico sin la intervención humana. Para conseguir esto, el sistema debe poseer redundancia **total o selectiva.**

Los sistemas con redundancia total utilizan unidades idénticas, operando simultáneamente con las originales para proteger al sistema contra fallos y/o errores. Cuando la redundancia es selectiva, se emplean procedimientos de recuperación en tiempo real para conmutar automáticamente, desde una unidad defectuosa a otra de reserva.

Más adelante se estudian algunas arquitecturas de sistemas con tolerancia a los fallos.

9.4.2.2. Flexibilidad

Indica la facilidad de un sistema para reconfigurar su topología y distribuir las responsabilidades de trabajo entre otros elementos.

9.4.2.3. Disponibilidad

La disponibilidad de un sistema multiprocesador indica cuanto tiempo está en funcionamiento dicho equipo, respecto de la duración total durante la que se hubiese deseado que funcionase. Se expresa en porcentaje de tiempo en el que funciona correctamente el sistema y se calcula según la fórmula:

$$Disponibilidad = \frac{MTBF}{MTBF + MTTR}$$

siendo MTBF el tiempo medio transcurrido antes del fallo y MTTR el tiempo medio necesario para reparar el fallo.

9.4.3. Aspectos del diseño y desarrollo de sistemas multiprocesadores

A continuación se describen algunas de las especificaciones que deben determinarse en el diseño, desarrollo o aplicación de sistemas multiprocesadores.

9.4.3.1. Descomposición de tareas

En aplicaciones complejas, es deseable dividir el trabajo principal en pequeñas tareas, lo más independientes que sea posible. Idealmente, cada tarea debería ser asignada a un procesador, reduciendo en la medida de lo posible la comunicación a nivel de datos entre procesadores.

9.4.3.2. Granularidad

La **granularidad** se define como el número de elementos de proceso que componen un computador paralelo.

Una **granularidad fina** consiste en el empleo de muchos elementos de proceso de poca potencia. En este caso, el grado de paralelismo es máximo. Los computadores de grano fino exigen técnicas de programación y algoritmos que potencien el paralelismo.

Una máquina de **grano grueso** consta de pocos elementos procesadores, pero de alta potencia cada uno.

9.5. Sistemas operativos para multiproceso

La interconexión entre los procesadores determinará las funciones de control del sistema operativo. El control será **centralizado** en el caso de organizaciones jerárquicas o verticales, y **distribuido** en organizaciones horizontales, en las que todos los procesadores son iguales desde el punto de vista lógico.

En un sistema con múltiples procesos concurrentes puede ocurrir que dos procesos traten de acceder a la vez a recursos que no pueden ser utilizados simultáneamente por más de un procesador (unidades de disco, I/O, etc.), o bien traten de acceder a recursos virtuales, como tablas dé datos o *buffers* de comunicación entre procesos. En estos casos, el encargado de proporcionar mecanismos para garantizar un uso exclusivo del recurso es el sistema operativo, y los procesos que deseen acceder al recurso deben competir por él. A esta exclusividad de acceso se le denomina **exclusión mutua entre procesos.**

9.5.1. Clasificación de los Sistemas Operativos para Multiproceso

Básicamente, se han utilizado tres tipos de organizaciones en el diseño de sistemas operativos para sistemas multiprocesadores:

1) **Configuración maestro-esclavo**.
2) **Supervisor independiente para cada procesador**.
3) **Supervisor flotante**.

Aunque muchos sistemas operativos no son ejemplos puros de alguna de las tres configuraciones anteriores, sino combinaciones de ellas.

9.5.1.1. Configuración maestro-esclavo

En esta configuración, la supervisión del sistema operativo siempre está a cargo de un solo procesador. Dicho procesador ejecuta el programa supervisor (**maestro**), que se encarga de mantener el estado de todos los procesadores que componen el sistema y de distribuirles el trabajo (**esclavos)**.

9.5.1.2. Supervisor independiente para cada procesador

Esta configuración es muy similar a la empleada en las redes de computadores, en las que cada procesador posee una copia del núcleo básico del programa supervisor, o *kernel.*

9.5.1.3. Supervisor flotante

En el caso de sistemas fuertemente acoplados hay varios procesadores similares que comparten memoria principal, dispositivos I/O, etc. En estos tipos de sistema no hay razón alguna que impida el manejo de los procesadores de forma simétrica. Como consecuencia aparece una tercera forma de organización de los sistemas operativos en la que todos los procesadores se tratan por igual a la hora de poder ejecutar el programa supervisor, ya que todos están igualmente dotados para ello.

Esta organización recibe el nombre de "supervisor flotante" porque en ella el programa supervisor puede *flotar*, cambiar de un procesador a otro. Esta es la organización más difícil de conseguir, pero también es la más flexible. Una de las principales ventajas de esta configuración es la mayor fiabilidad del sistema.

9.6. Explotación de la concurrencia

9.6.1. Características de la Concurrencia

Los procesos concurrentes tienen las siguientes características:

Interacción entre procesos: Los programas concurrentes implican interacción entre los distintos procesos que los componen:

- Los procesos que comparten recursos y compiten por el acceso a los mismos.
- Los procesos que se comunican entre sí para intercambiar datos.

En ambas situaciones se necesita que los procesos sincronicen su ejecución para evitar conflictos, o establecer contacto para el intercambio de datos. La interacción entre procesos se logra mediante variables compartidas o bien mediante el paso de mensajes.

Gestión de recursos: Los recursos compartidos necesitan una gestión especial. Un proceso que desee utilizar un recurso compartido debe solicitar dicho recurso, esperar a adquirirlo, utilizarlo y después liberarlo. Si el proceso solicita el recurso pero no puede adquirirlo en ese momento, es suspendido hasta que el recurso esté disponible. La gestión de recursos compartidos es problemática y se debe realizar de tal forma que se eviten situaciones de retraso indefinido (espera indefinidamente por un recurso) y de deadlock (bloqueo indefinido o abrazo mortal).

Comunicación: La comunicación entre procesos puede ser síncrona, cuando los procesos necesitan sincronizarse para intercambiar los datos, o asíncrona, cuando un proceso que suministra los datos no necesita esperar a que el proceso receptor los recoja, ya que los deja en un buffer de comunicación temporal.

9.6.2 Problemas de la concurrencia.

Los programas concurrentes, a diferencia de los programas secuenciales, tienen una serie de problemas muy particulares:

Violación de la exclusión mutua: En ocasiones puede ocurrir que ciertas acciones que se realizan en un programa concurrente no proporcione los resultados deseados. Esto se debe a que una parte del programa donde se realizan dichas acciones constituye una región crítica, es decir, es una parte del programa en la que se debe garantizar que si un proceso accede a la misma, ningún otro podrá acceder. Se necesita pues garantizar la exclusión mutua.

Abrazo mortal o Deadlock: También se le denomina bloqueo mutuo. Cuando se realiza una ejecución concurrente puede ocurrir que un proceso deba esperar hasta que se produzca un determinado evento. Si este evento se produce, el proceso es "despertado" y se reanuda la ejecución sin ningún problema. Pero si dicho evento no se produce, el proceso permanecerá bloqueado para siempre, en estado de *deadlock* o *abrazo mortal*.

Para aclarar este concepto utilizaremos un ejemplo basado en un sistema con dos procesos P1 y P2, y dos recursos R1 y R2. Supóngase que el proceso P1 obtiene el recurso R1 y el proceso P2 obtiene el recurso R2. A continuación, P1 solicita una petición de acceso a R2. Como el recurso R2 todavía está asignado a P2, el proceso P1 se bloquea hasta que el recurso quede libre R2. Si en estas circunstancias el proceso P2 realiza una petición de acceso al recurso R1, dado que R1 aún está asignado, el proceso P2 se bloquea hasta que quede libre dicho recurso. Como resultado tenemos a los procesos P1 y P2 bloqueados, cada uno de ellos esperando a que se les asigne el recurso que esperan, pero que tiene asignado el otro. En este caso se dice que P1 y P2 se encuentran mortalmente abrazados.

Desgraciadamente, no todos los abrazos mortales son tan sencillos como el del ejemplo anterior, pudiendo surgir casos más complejos cuando existen muchos procesos y muchos recursos.

El abrazo mortal se mantiene hasta que se produzca una "acción especial" por parte de una fuerza externa, tal como el operador o el sistema operativo. Este efecto se puede producir en la comunicación de procesos y más frecuentemente en la gestión de recursos. Existen cuatro condiciones necesarias para que se pueda producir:

1) Los procesos necesitan acceso exclusivo a los recursos.
2) Los procesos necesitan mantener ciertos recursos exclusivos mientras esperan por otros.
3) Los recursos no se pueden obtener de los procesos que están a la espera.
4) Existe una cadena circular de procesos en la cual cada proceso posee uno o más de los recursos que necesita el siguiente proceso en la cadena.

Retraso indefinido o starvation: Un proceso se encuentra en *starvation* si es retrasado indefinidamente esperando un suceso que puede no ocurrir nunca. Esta situación se puede producir si la gestión de recursos emplea un algoritmo en el que no se tenga en cuenta el tiempo de espera del proceso.

Espera ocupada: En ocasiones cuando un proceso se encuentra a la espera por un suceso, una forma de comprobar si el suceso se ha producido es verificando continuamente si el mismo se ha realizado ya. Esta solución de espera ocupada es muy poco efectiva, porque desperdicia tiempo de procesamiento, y se debe evitar. La solución ideal es suspender el proceso y continuar cuando se haya cumplido la condición de espera.

9.6.3. Especificación de la Concurrencia

Conviene disponer de una notación clara que exprese las operaciones concurrentes. Se denominan procesos concurrentes a aquellos cuya ejecución se solapa en el tiempo. Más concretamente, podemos decir que: *dos procesos son concurrentes si la primera operación de uno de ellos comienza antes de que termine la última operación del otro*.

El programa paralelo de un sistema multiprocesador estará formado por dos o más procesos relacionados entre sí. Para comprender un programa paralelo, la clave es identificar los procesos y los objetos que comparten.

Existen dos formas de diseñar programas paralelos. La primera consiste en indicar la concurrencia de **forma explícita**, por lo que el programador ha de expresar la concurrencia en el propio programa. La otra consiste en indicar la concurrencia de **forma implícita,** siendo aquí el compilador quien determina qué es lo que se ha de ejecutar en paralelo. Este segundo procedimiento es más apropiado para arquitecturas de flujo de datos (*Data Flow Architectures*).

A continuación se muestras varias alternativas para expresar el paralelismo de manera explícita.

9.6.3.1. Declaración FORK-JOIN

En este caso se dispone de dos declaraciones para su uso en un lenguaje de alto nivel. La declaración FORK indica la creación de un nuevo proceso y la declaración JOIN espera a que termine un proceso previamente creado. Un sencillo ejemplo de aplicación de las declaraciones FORK-JOIN es el siguiente:

```
Programa P1;         Programa P2;
.............        .............
FORK P2;             .............
.............        .............
JOIN P2;             END P2;
.............
END P1;
```

Como vemos en el código, la ejecución de P2 se inicia al ejecutarse FORK P2 en el programa P1. A partir de este punto, P1 y P2 se ejecutan concurrentemente hasta que P1 pretende ejecutar la declaración JOIN P2, momento en que se detiene la ejecución de P1 hasta que P2 termina. Una vez concluido P2, se ejecuta la declaración JOIN P2 del proceso P1, y la ejecución de P1 se traslada a las instrucciones siguientes al JOIN P2.

9.6.3.2. Declaración COBEGIN-COEND

La declaración **cobegin-coend** es una forma "estructurada" de indicar la ejecución concurrente de un conjunto de declaraciones.

Véase cómo en el siguiente fragmento de código, cada proceso o declaración del conjunto $S_1, S_2, \ldots, S_n$ puede ejecutarse concurrentemente mediante la siguiente construcción:

```
begin
   S0;
      cobegin S1; S2; ...; Sn; coend
   Sn+1
end
```

Cada una de las S_i puede ser cualquier declaración, incluyendo una cobegin-coend o un bloque de declaraciones locales. De esta forma, la declaración cobegin-coend indica explícitamente qué partes de un programa pueden ejecutarse concurrentemente. La figura 9.8 muestra gráficamente la concurrencia correspondiente al fragmento de código anterior. En este caso, las declaraciones situadas entre cobegin y coend se ejecutan concurrentemente, pero sólo después de que la declaración S_0 se haya ejecutado. Por otro lado, la declaración S_{n+1}, se ejecuta sólo después de que las ejecuciones de $S_1, S_2, \ldots, S_n$ hayan finalizado.

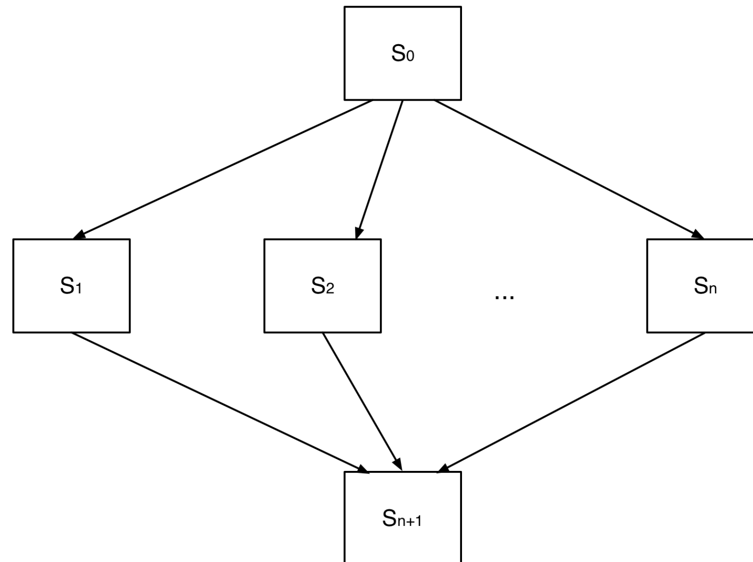


figura 9.8. Expresión gráfica de la concurrencia del ejemplo.

Las declaraciones concurrentes pueden ser anidadas de forma arbitraria como en el siguiente ejemplo, cuyo gráfico se muestra en la figura 9.9.

```
begin
    S0;
        cobegin
            S1;
                begin
                    S2;
                        cobegin S3; S4; S5; coend
                    S6;
                end
            S7;
        coend
    S8;
end
```

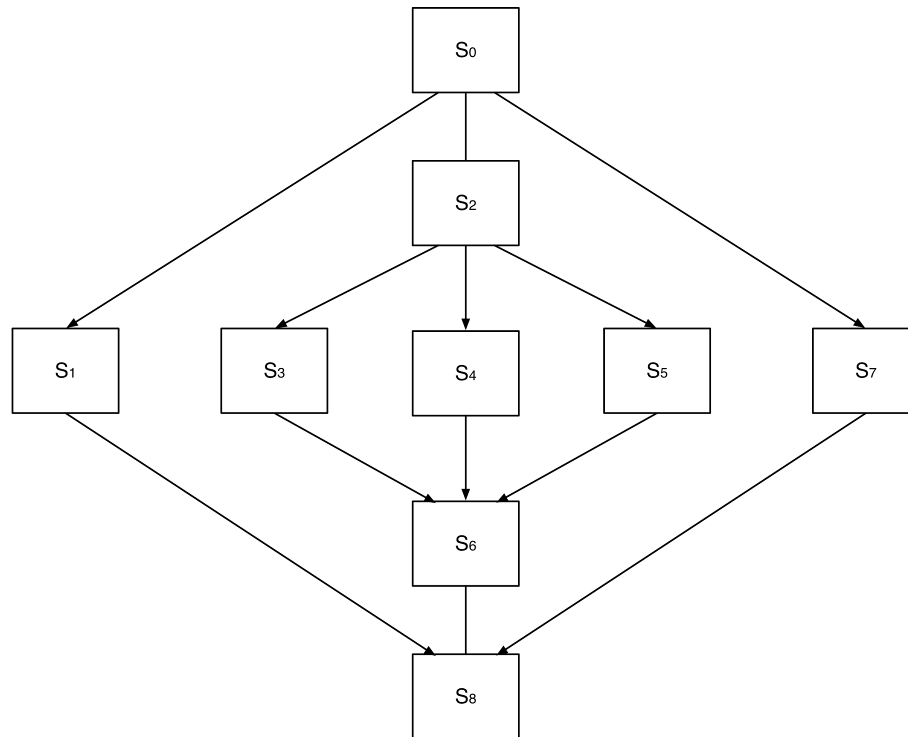


figura 9.9 Expresión gráfica de la concurrencia del ejemplo.

9.6.4. Detección de paralelismo en los programas

En el punto anterior se han analizado un par de alternativas mediante las cuales el programador puede expresar el paralelismo de forma explícita. Sin embargo, en algunos casos es posible ceder esta tarea al compilador, para detectar el paralelismo independientemente del programador. De esta forma, el compilador reconoce automáticamente, a partir del código fuente, los procesos que pueden ejecutarse en paralelo (paralelismo implícito).

La dependencia de datos es el factor principal usado en la detección de paralelismo entre procesos.

Para que dos programas se puedan ejecutar en paralelo se deben verificar ciertas condiciones, las condiciones de Bernstein, que se presentan a continuación:

1) Sea I_i el conjunto de todas las variables de entrada necesarias para ejecutar P_i.

2) Sea O_i el conjunto de todas las variables de salida generadas por el proceso P_i.

Las condiciones de Bernstein para dos procesos P_1 y P_2 son las siguientes:

- $I_1 \cap O_2 = \varnothing$
- $I_2 \cap O_1 = \varnothing$
- $O_1 \cap O_2 = \varnothing$

Si se cumplen las tres condiciones entonces se dice que P_1 y P_2 pueden ser ejecutados en paralelo y se denota como $P_1 \parallel P_2$. Esta relación de paralelismo es conmutativa ($P_1 \parallel P_2$ ==> $P_2 \parallel P_1$) pero no es transitiva ($P_1 \parallel P_2$ y $P_2 \parallel P_3$ =/=> $P_1 \parallel P_3$).

Las condiciones de Bernstein aunque definidas para procesos son extrapolables a nivel de instrucciones.

EJEMPLO

Sean las tareas T1, T2, y T3, que representan la realización de las siguientes operaciones aritméticas con matrices:

$$T_1; \quad X \leftarrow (A+B)*(A-B)$$

$$T_2; \quad Y \leftarrow (C-D)*(C+D)^{-1}$$

$$T_3; \quad Z \leftarrow X+Y$$

En este ejemplo tenemos que $I_1 = (A,B)$, $I_2 = (C,D)$, $O_1 = (X)$ y $O_2 = (Y)$. Puesto que $I_1 \cap O_2 = \varnothing$, $I_2 \cap O_1 = \varnothing$ y $O_1 \cap O_2 = \varnothing$, las tareas T1 y T2 pueden ejecutarse en paralelo. Como $I_3 = (X,Y)$, resulta ser $I_3 \cap O_1 \neq \varnothing$ y $I_3 \cap O_1 \neq \varnothing$, por lo que T3 no puede ejecutarse en paralelo con T1 ni con T2.

De esto se deduce que es posible ejecutar T1 y T2 de forma concurrente y, posteriormente, ejecutar T3 tal como expresa el código siguiente:

```
begin
   cobegin
      X ← (A + B) * (A − B);
      Y ← (C − D) * (C + D)^-1;
   coend
   Z ← X + Y
end
```

9.7. Mecanismos de comunicación entre procesos

En muchas ocasiones es necesario que dos procesos se comuniquen entre sí, o bien sincronicen su ejecución de tal forma que uno espere al otro, o sincronicen su acceso a un recurso compartido.

En ocasiones los procesos comparten un espacio de almacenamiento común (un fichero, una zona de memoria, etc.) en la que cada uno puede leer o escribir. Los problemas surgen cuando el acceso a dicha zona no está organizado, sino que es más o menos aleatorio o más bien dependiente de las condiciones de la máquina.

A estas situaciones se las denomina condiciones de competencia, y se deben solucionar *garantizando que sólo un proceso al mismo tiempo accede a la zona compartida*. El objetivo es garantizar la exclusión mutua, una forma de garantizar que si un proceso utiliza una variable, archivo o cualquier otro objeto compartido, los demás procesos no podrán utilizarlo.

Un problema muy común se presenta cuando dos o más procesos concurrentes comparten datos que pueden ser modificados. En este caso, si a un proceso se le permite acceder a un conjunto de variables que están siendo actualizadas por otro proceso, pueden producirse resultados erróneos en las operaciones. Por lo tanto, hay que disponer de mecanismos de control del acceso a las variables compartidas para garantizar que sólo un proceso tendrá acceso exclusivo a las secciones de programa y datos que puedan ser modificados. Tales segmentos se denominan secciones críticas, es decir, *una sección crítica es una secuencia de declaraciones que debe ejecutarse como una operación indivisible*.

La condición de la exclusión mutua no es suficiente para evitar todos los conflictos que se pueden producir entre procesos paralelos que cooperan y emplean datos compartidos. Existen varias condiciones que se deben cumplir para obtener una buena solución:

- *Garantizar la exclusión mutua*: Dos procesos no deben encontrarse al mismo tiempo en su sección crítica.
- *Indeterminación*: No se deben hacer hipótesis acerca de la velocidad o el número de procesadores, durante la ejecución de los procesos. Ninguno de los procesos que se encuentran fuera de su sección crítica puede bloquear a otros procesos.
- *Evitar el retraso indefinido*: Ningún proceso debe esperar eternamente a entrar en su región crítica.

Un proceso se ejecuta con una velocidad impredecible y genera acciones o sucesos que deben ser reconocidos por otros procesos que cooperan con él. El conjunto de restricciones que suponen estas acciones o sucesos constituyen la sincronización, requerida para la cooperación entre procesos. Los mecanismos de sincronización se utilizan para retardar la ejecución de un proceso de forma que se satisfagan las restricciones propuestas.

La comunicación y sincronización entre procesos se realiza a través de uno de los dos modelos siguientes:

1) Utilización de variables compartidas.
2) Técnica de intercambio de mensajes o "*message passing*".

A continuación se describen algunas primitivas de sincronización, tales como espera ocupada (*busy-waiting*), semáforos, regiones críticas condicionales, monitores, etc.

9.7.2. Ocupado-Esperando (*Busy-Waiting*)

Una manera de llevar a cabo la sincronización consiste en poner una variable compartida, una cerradura, a 1 cuando se va a entrar en la región crítica, y devolverla al valor 0 a la salida. Esta solución en sí misma no es válida porque la propia cerradura es una variable crítica. La cerradura puede estar a 0 y ser comprobada por un proceso A, éste ser suspendido, mientras un proceso B chequea la cerradura, la pone a 1 y puede entrar a su región crítica; a continuación A la pone a 1 también, y tanto A como B se pueden encontrar en su sección crítica al mismo tiempo. Existen muchos intentos de solución a este problema:

- El algoritmo de Dekker
- El algoritmo de Peterson
- La instrucción hardware TSL (Test & Set Lock): Instrucción de prueba y activación.

Las soluciones de Dekker, Peterson y TSL son correctas pero emplean espera ocupada. Básicamente lo que ocurre es que cuando un proceso desea entrar en su sección crítica comprueba si está permitida la entrada o no. Si no está permitida, el proceso se queda en un bucle de espera hasta que se consigue el permiso de acceso.

Como el proceso debe esperar a que se cumpla la condición deseada y debe comprobar el valor de la variable hasta que tenga el previsto, a esta técnica se la denomina con el nombre de espera ocupada o busy-waiting.

Esto produce un gran desperdicio de tiempo de CPU y pueden aparecer otros problemas como la espera indefinida.

Una solución más adecuada es la de bloquear o dormir el proceso (SLEEP) cuando está a la espera de un determinado evento, y despertarlo (WAKEUP) cuando se produce dicho evento. Esta idea es la que emplean las siguientes soluciones.

9.7.3. Semáforos

Dijkstra fue uno de los primeros en darse cuenta de la dificultad de usar mecanismos de bajo nivel para sincronizar procesos. Este inconveniente propició el desarrollo de dos operaciones, **P** y **V**, que pueden compartirse por varios procesos y que logran la sincronización por exclusión mutua de una manera muy eficiente.

Las operaciones P y V se denominan "primitivas" y se suponen indivisibles. Estas primitivas actúan sobre variables comunes especiales, denominadas **semáforos**.

Se define un semáforo como *una variable entera de valor no negativo sobre la que se definen las operaciones P y V*. Así, la operación P se utiliza para solicitar permiso para entrar en la sección crítica, mientras que la operación V registra la conclusión del acceso a dicha sección. Los mecanismos de estas primitivas pueden definirse de la siguiente forma:

```
P(Semáforo s)
{
      if(s>0)
          s = s-1;
      else
          wait();
}


V(Semáforo s)
{
      if(!procesos_bloqueados)
          s = s+1;
      else
          signal();
}
```

donde WAIT es una primitiva que bloquea el proceso y SIGNAL otra que comprueba la lista de procesos en espera, seleccionando uno y poniéndolo en estado listo para ejecutarse.

Cuando un semáforo S toma únicamente los valores 0 y 1, se dice que el semáforo es binario. En este caso el semáforo actúa como un bit de bloqueo que permite un solo acceso a la sección crítica en un instante dado. Si, de otra forma, el semáforo S puede tomar cualquier valor entero, se le denomina "semáforo de cuenta". Estos semáforos se usan a menudo para sincronizar condiciones que controlan la asignación de recursos. Para ello, el semáforo toma como valor el número de unidades de recursos que existen. En este caso, una operación P se utilizaría para poner el proceso en espera hasta que se libere una unidad del recurso, mientras que la operación V se ejecutaría para liberar la unidad del recurso. Los semáforos binarios son suficientes para establecer algunos tipos de sincronización, sobre todo en el caso de que los recursos tengan solamente una unidad.

A continuación, el siguiente programa ofrece una solución para la exclusión mutua de dos procesos, en términos de semáforos:

```
Program Mutex__Example
   var mutex: semaphore initial (1);
   process P1;
      loop
            P (mutex); <protocolo de entrada>
            Sección crítica;
            V (mutex); <protocolo de salida>
            Sección no crítica;
         end
      end
      process P2;
         loop
            P (mutex); <protocolo de entrada>
            Sección crítica;
            V (mutex);
            Sección no crítica;
         end
      end
end
```

La utilización en este ejemplo de las primitivas P y V asegura la exclusión mutua y la ausencia de abrazo mortal.

Los semáforos normalmente se crean a través del núcleo (o kernel) del sistema operativo. En todo momento, un proceso está, bien listo para ejecutarse sobre un procesador, o bien bloqueado en espera de que algún otro complete una operación P. El kernel mantiene una lista de procesos preparados, y reparte el tiempo de ejecución del procesador o procesadores entre estos procesos. Los descriptores para procesos que están en cola de espera se mantienen en una cola asociada al semáforo que atienden. La ejecución de una operación P ó V hará que se ejecute una rutina de servicio del kernel. Para una operación P, y si el semáforo tiene valor positivo, se decrementa el semáforo en una unidad y el descriptor correspondiente al proceso que va a acceder a la sección crítica se coloca en la cola de espera asocicada a dicho semáforo. Para una operación V, si la cola de espera asociada al semáforo no está vacía, se coloca el descriptor correspondiente en la cola de la lista de procesos preparados y se incrementa el semáforo en una unidad.

9.7.4. Regiones Críticas Condicionales y Monitores

Aunque los semáforos pueden utilizarse para programar casi todos los tipos de sincronización, las primitivas P y V son primitivas desestructuradas, y por ello es fácil cometer un error cuando se utilizan. El acceso a una sección crítica debe comenzar con una operación P y concluir con una V sobre el mismo semáforo. Si por error se omiten P ó V, o accidentalmente se realiza la operación P en un semáforo y la operación V sobre otro, pueden producirse efectos desastrosos al no poderse asegurar, con certeza, la ejecución mutuamente exclusiva.

Las regiones críticas condicionales y los monitores son mecanismos de sincronización, consecuencia de los semáforos, que proporcionan formas estructuradas de control de acceso a las variables compartidas.

Regiones críticas condicionales: Solución propuesta por Hoare y Brinch Hansen como mejora de los semáforos. Consiste en definir las variables de una región crítica como recursos con un nombre, de esta forma la sección crítica se precede con el nombre del recurso que se necesita y opcionalmente una condición que se debe cumplir para acceder a la misma. Es un buen mecanismo, pero no suele estar soportado por la mayoría de los lenguajes de programación.

Monitores: Esta solución también fue propuesta por Brinch Hansen y Hoare. Un monitor es un conjunto de procedimientos, variables y estructuras de datos que se agrupan en un determinado módulo. Los procesos pueden llamar a los procedimientos del monitor cuando lo deseen para realizar las operaciones sobre los datos compartidos, pero no pueden acceder directamente a las estructuras de datos internas del monitor. Su principal propiedad para conseguir la exclusión mutua es que sólo un proceso puede estar activo en un monitor en cada momento.

9.7.5. Sincronización basada en intercambio de mensajes (*Message Passing*)

El paso de mensajes o transferencia de mensajes es el modelo más empleado para la comunicación entre procesos. Puede contemplarse como una extensión de los semáforos para transmitir datos y a la vez llevar a cabo la sincronización. Cuando se usa ésta técnica, los procesos envían y reciben mensajes en lugar de leer y escribir en variables compartidas. La comunicación se lleva a cabo como consecuencia de que un proceso, sobre la recepción de un mensaje, obtiene valores de algún proceso emisor. La sincronización se consigue por el mero evento de recepción del mensaje. Empleando mensajes se pueden implementar semáforos y monitores.

Para la comunicación se emplean las primitivas SEND (para enviar un mensaje) y RECEIVE (para poner al proceso a la espera de un mensaje).

La designación más sencilla de un canal de comunicación se conoce como **designación directa** (direct naming). En este caso, la ejecución de:

SEND ‘card’ to ‘executer’

envía un mensaje cuyos datos son los valores que contiene la expresión ‘card’. Este mensaje sólo podrá ser recibido por ‘executer’. De forma similar, la ejecución de:

RECEIVE ‘line’ from ‘executer’

recoge la información del mensaje. El mensaje debe proceder del proceso ‘executer’ y, por tanto, otorga control sobre la procedencia del mensaje. La recepción de un mensaje provoca siempre la destrucción del mensaje.

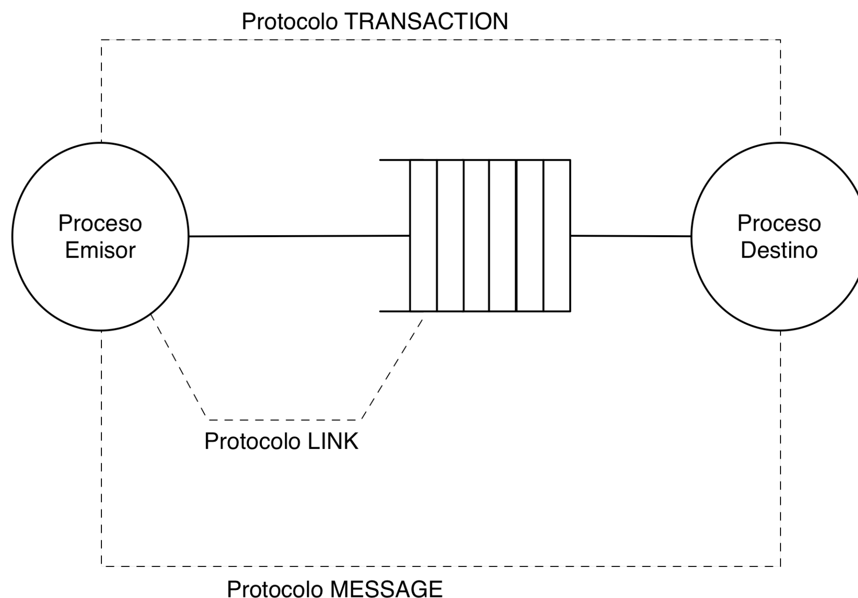


figura 9.10. Representación esquemática del protocolo de comunicación.

El envío del mensaje puede ser reconocido: a) después de situarse en un buffer de recepción (protocolo de comunicación LINK), b) tras la recepción por parte del proceso receptor (verifica el comienzo de la recepción del mensaje (protocolo MESSAGE)), o bien c) una vez que el proceso receptor haya efectuado alguna operación con el mensaje (protocolo TRANSACTION).

Un ejemplo de relieve que muestra la interacción entre procesos es la relación cliente/servidor. Algunos procesos servidores prestan servicio a ciertos procesos clientes. Un cliente puede solicitar que se realice un servicio mediante el envío de un mensaje a uno de los servidores. Un servidor recibe, repetidamente, peticiones de servicio de un cliente, ejecuta el servicio y, si es necesario, envía un mensaje de finalización a dicho cliente. Sin embargo, este ejemplo también sirve para mostrar que la utilización del canal de comunicación según designación directa no siempre se adapta a la interacción cliente/servidor, ya que el RECEIVE de un servidor debe permitir la recepción de un mensaje enviado por cualquier cliente. Si sólo hay un cliente, la designación directa es muy eficiente, sin embargo las dificultades aparecen cuando hay más de un cliente debido a que, por lo menos, debe haber un RECEIVE para cada uno. De la misma forma, si hay más de un servidor, y todos son idénticos, el SEND de un cliente debe producir un mensaje que pueda ser recibido por cualquier servidor. Esto no es sencillo de implementar con la designación directa, será necesario un modelo más sofisticado para definir los canales de comunicación.

Otro posible (y frecuente) método es el que se basa en la utilización de nombres globales, denominados **buzones** (*mailboxes*). Un buzón puede aparecer como destinatario en cualquier declaración SEND de un proceso y como fuente en cualquier declaración RECEIVE. Por ello, los mensajes enviados a un buzón determinado puede ser recibidos por cualquier proceso que ejecute una operación RECEIVE sobre dicho buzón. Este modelo se adapta muy bien a la

programación de interacciones clientes/servidores (clientes que envían sus peticiones de servicio a un buzón, donde las reciben los servidores).

La forma más sencilla de implementar un buzón es aquella en la que el nombre de un buzón puede aparecer como la fuente de una declaración RECEIVE en un solo proceso. A este tipo de buzones se les denomina **puertos**.

9.7.5.1. Sincronización en el intercambio de mensajes

Una propiedad importante del intercambio de mensajes hace referencia a que su ejecución puede causar un retardo (en función del protocolo de comunicación, figura 9.10). Una declaración se denomina **no bloquente** (nonblocking) si su ejecución no produce retardo en el proceso invocado; en caso contrario, se denomina **bloqueante** (blocking).

En algunos modelos de paso de mensajes, los mensajes se almacenan en un buffer desde que se mandan hasta que se reciben. Si el buffer está lleno al ejecutar una declaración SEND, pueden producirse dos alternativas: bien se retarda el envío hasta que haya espacio en el buffer para guardar el mensaje, o bien se comunica al proceso invocador que el mensaje no puede enviarse por estar lleno el buffer. De forma análoga, si en la ejecución de una declaración RECEIVE no hay disponible ningún mensaje, puede producirse una espera hasta tener uno dispobible, o bien terminar, señalando que no hay disponible mensaje alguno.

Si un sistema dispone de un buffer de capacidad ilimitada, no se retardará proceso alguno cuando se ejecute una declaración SEND. A este procedimiento se denomina **intercambio de mensajes asíncrono**, y también SEND no retardado. Por otro lado, cuando no hay almacenamiento en el buffer, la ejecución de la declaración SEND se retarda hasta que se ejecute la declaración RECEIVE correspondiente. Este caso recibe el nombre de **intercambio de mensajes síncrono**. Cuando se usa este último procedimiento, un intercambio de mensaje también supone un punto de sincronización entre emisor y el receptor. Entre ambos extremos se encuentra el intercambio de mensajes con buffer, en el que el buffer tiene una capacidad limitada.

9.8. Algoritmos paralelos para sistemas multiprocesadores

9.8.1. Clasificación de los algoritmos paralelos

Un algoritmo paralelo destinado a un sistema multiprocesador es un conjunto de k procesos concurrentes que pueden trabajar simultáneamente y cooperando entre ellos para resolver un determinado problema. En el caso de que k = 1, se le denomina algoritmo secuencial.

Para que un algoritmo paralelo trabaje de forma correcta y efectiva para solucionar un problema, existirán puntos en los que los procesos requieran sincronizarse o comunicarse entre sí. Éstos se denominan *puntos de interacción*.

Debido a las interacciones entre procesos, algunos procesos pueden quedar bloqueados en ciertos instantes. Los algoritmos paralelos en los que ciertos procesos deben esperar a otros se denominan *algoritmos paralelos sincronizados*. Puesto que el tiempo de ejecución varía de un proceso a otro, los procesos que tienen que sincronizarse en un punto fijo deben esperar al que más tarda en llegar. Esta circunstancia puede conducir a una peor utilización de los procesadores de lo que cabría esperar.

Para solucionar los problemas de los algoritmos paralelos sincronizados, pueden utilizarse *algoritmos paralelos asíncronos*, en los cuales no se necesita que los procesos se esperen entre sí, y la comunicación entre ellos se desarrolla a través de variables compartidas almacenadas en memoria común, y actualizadas dinámicamente. De todas formas, al utilizar memoria común también pueden surgir retrasos de menor orden originados por conflictos de acceso a las variables comunes.

Otra alternativa para construir algoritmos paralelos es la técnica denominada *macrosegmentación* (*macropipelining*), que es aplicable si la ejecución puede ser dividida en partes, denominadas etapas, de tal forma que la salida de una o varias partes es la entrada de otra. La figura 9.11 muestra el flujo de un esquema de programa macrosegmentado. Como vemos, se trata de una tipo de algoritmo paralelo asíncrono que utiliza buffers para desacoplar el funcionamiento de las diferentes etapas.

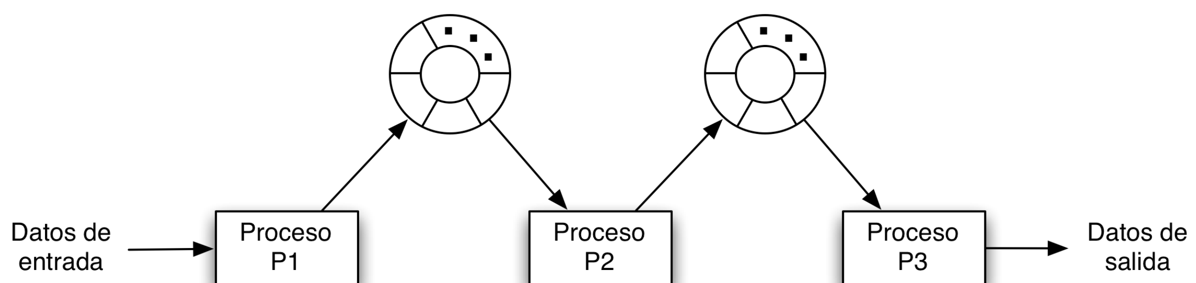


figura 9.11. Flujo de Programa para la macrosegmentación.

EJEMPLO

Veamos a continuación un ejemplo de realización de un algoritmo sincronizado. Como sabemos, los algoritmos sincronizados están formados por varios procesos, existiendo alguno de ellos en el que alguna de sus etapas no se activa hasta que otro haya concluido cierta etapa concreta.

La sincronización puede realizarse usando alguna de las diversas primitivas de sincronización consideradas anteriormente.

El ejemplo que nos ocupa trata del cálculo de la matriz

$$Z = A \cdot B + ((C + D) \cdot (I + G))$$

utilizando una descomposición de ésta en diferentes procesos que puedan ser distribuidos en un sistema multiprocesador.

Se puede diseñar un algoritmo sincronizado creando tres procesos P1, P2 y P3, tal como se muestra en la figura 9.12. Los procesos P1, P2 y P3 constan de dos, una y dos etapas, respectivamente. El código de la implementación se ilustra en la figura 9.13.

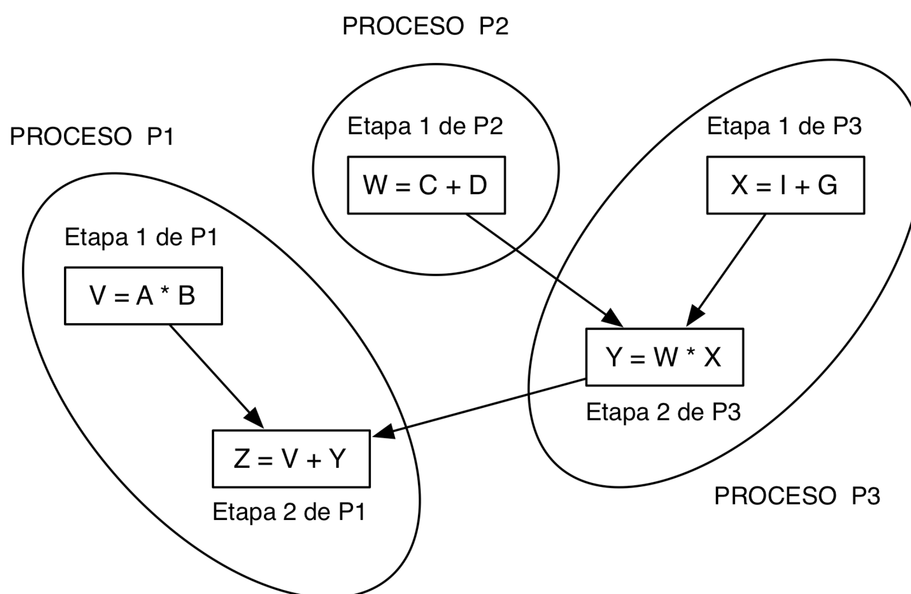


figura 9.12. Flujo de Programa para la macrosegmentación.

```
var w, y: shared real;
var Sw, Sy: semaphore; initial Sw = Sy = 0;
   cobegin
      process P1: begin
               V ← A*B; // etapa 1 de P1 //
               P (Sy);
               Z ← V + Y; // etapa 2 de P1 //
            end
      process P2: begin
               W ← C + D; // etapa 1 de P2 //
               V (Sw)
            end
      process P3: begin
               X ← I + G; // etapa 1 de P3 //
               P (Sw);
               Y ← W*X; // etapa 2 de P3 //
               V (Sy);
               end
coend
```

figura 9.13. Código de la implementación.

9.9. Arquitecturas tolerantes a fallos

9.9.1. Introducción

Desde la primera generación de computadores, al comienzo de la década de los 40, una de las cuestiones más importantes es la de incrementar su fiabilidad. Cuando éstas se utilizan en misiones espaciales, sistemas biomédicos, etc, un pequeño error puede producir la pérdida de una vida o de grandes cantidades de dinero y años de investigación. En estas condiciones es fundamental diseñar computadores que tengan la capacidad de operar correctamente a pesar de que se produzca un fallo en el equipo físico o en el sistema lógico. A este tipo sistemas se les denomina **tolerantes a fallos**.

La *tolerancia a fallos (fault-tolerance)* se puede definir como *la capacidad de un sistema computador de ejecutar correctamente unos algoritmos, aunque se produzcan errores en el hardware o en el software.*

9.9.2. Técnicas de redundancia en el equipo físico

La **redundancia hardware** se define como el uso de componentes hardware adicionales que permitan continuar operando correctamente al sistema, incluso en el caso de fallos en el equipo físico.

El coste que supone la utilización de componentes adicionales para crear un sistema con redundancia hardware supone un fuerte argumento contra el uso de la redundancia. Sin embargo,

la constante reducción de precios de los componentes hardware ha facilitado la utilización de tales sistemas.

Las técnicas básicas de redundancia hardware son:

1) Redundancia estática
2) Redundancia dinámica
3) Redundancia híbrida

9.9.2.1. Redundancia estática

La **redundancia estática** (también llamada redundancia masiva o enmascaramiento) se basa en la utilización de componentes adicionales de forma que el efecto de un fallo en un circuito, componente, subsistema, señal o programa, se enmascara inmediatamente por los circuitos similares que operan concurrentemente y que se hallan conectados permanentemente.

La técnica de redundancia estática más simple es la llamada **redundancia modular triple** (TMR), que consiste en usar tres módulos idénticos funcionando en paralelo, figura 9.14. La salida de cada uno de los tres módulos se lleva a la entrada de un selector de mayoría (Voter), cuya salida se obtiene de la salida que sea mayoría de entre las salidas de los módulos. La redundancia modular triple puede extenderse para incluir N módulos redundantes funcionando según el mismo principio, obteniendo así la redundancia modular de orden N (NMR), donde N siempre debe ser un número impar. Un sistema con redundancia modular de orden N puede admitir hasta (N-1)/2 fallos.

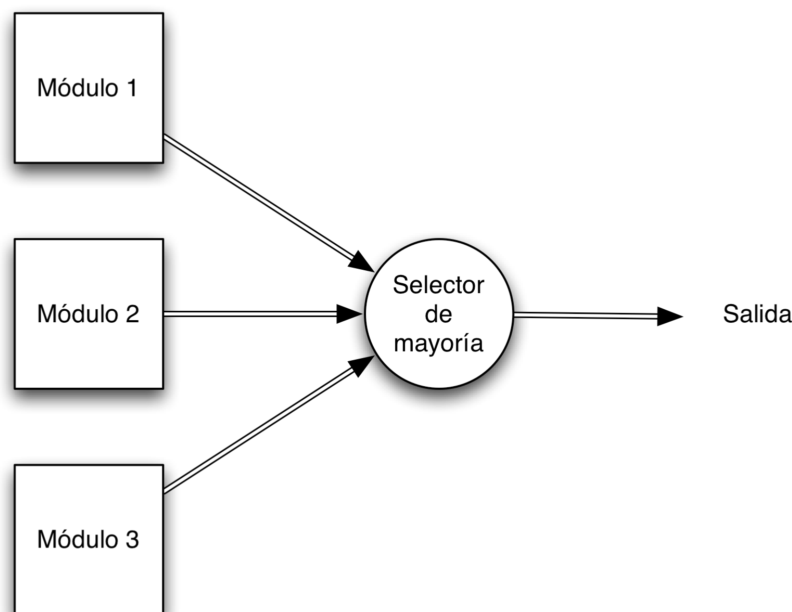


figura 9.14. Redundancia Modular Triple.

Las ventajas más importantes de la redundancia estática son:

1) La corrección del fallo es inmediata. Además, el fallo en un módulo nunca afecta al circuito.

2) No se requieren procedimientos para detectar fallos.

3) La conversión de un sistema no redundante a un sistema con redundancia estática es relativamente fácil de realizar, basta con incluir unidades idénticas de cada módulo redundante.

9.9.2.2. Redundancia dinámica

En este tipo de redundancia sólo opera una unidad, existiendo una o varias unidades de reserva esperando para reemplazarla si se produce un fallo. Por tanto, es necesario un método para conectar al circuito la unidad de reserva que va a sustituir a la que ha fallado y además un procedimiento para la detección del fallo en el circuito.

Existen dos formas de realizar la conexión de la unidad de reserva al al circuito:

- Todas las unidades de reserva están "encendidas" y funcionando a la vez, de forma que cuando se produce un fallo se conecta la salida de una de las unidades de reserva al circuito y se desconecta la unidad que ha fallado.

- Sólo está "encendida" la unidad que opera en el circuito. Cuando se produce un fallo, dicha unidad se "apaga", y una de las unidades de reserva se "enciende" y se conecta al circuito. De esta forma, se logra que las unidades de reserva no consuman energía.

En este tipo de redundancia el procedimiento para detectar el fallo es bastante más complicado que en la redundancia estática, y puede implicar la utilización de técnicas concurrentes o periódicas, es decir, la utilización de señales redundantes y rutinas especiales de diagnóstico para la comprobación de ciertos puntos del sistema.

Las principales ventajas de la redundancia dinámica son:

1) Pueden ser utilizadas todas las unidades de reserva y, por tanto, el sistema admite tantos fallos como unidades de reserva.

2) La fiabilidad puede mejorarse añadiendo unidades de reserva, sin que esto afecte al sistema

9.9.2.3. Redundancia híbrida

La redundancia híbrida es una combinación de redundancia estática y dinámica. El esquema está formado, básicamente, por un sistema con redundancia modular triple más varias unidades de reserva. Por una parte, la redundancia estática se emplea para detectar el fallo. Cuando se

detecta un fallo, se utiliza el conmutador para desconectar la unidad que ha fallado y sustituirla por cualquiera de las unidades de reserva restantes, tal como se muestra en la figura 9.15. A este esquema se le denomina sistema híbrido (3, S), donde S es el número de unidades de reserva.

Cuando se produce un fallo en el sistema de la figura 9.15 (la salida de un módulo no coincide con la de los otros dos), se desconecta el módulo que falla y se conecta el de reserva. De esta forma, se obtiene un sistema híbrido (3, S-1). En caso de S fallos sucesivos será necesario utilizar todos los módulos de reserva, por lo que finalmente se tendrá un sistema híbrido (3, 0) o, lo que es lo mismo, un sistema con redundancia modular triple.

La redundancia híbrida posee todas las ventajas de la redundancia dinámica y evita su principal desventaja, el problema de la construcción de complejos procedimientos de detección de fallos. Por otra parte, el circuito selector de mayoría enmascara el efecto de cualquier fallo. Sin embargo, el circuito de conmutación es más complejo que en el caso de un sistema con redundancia estática o dinámica.

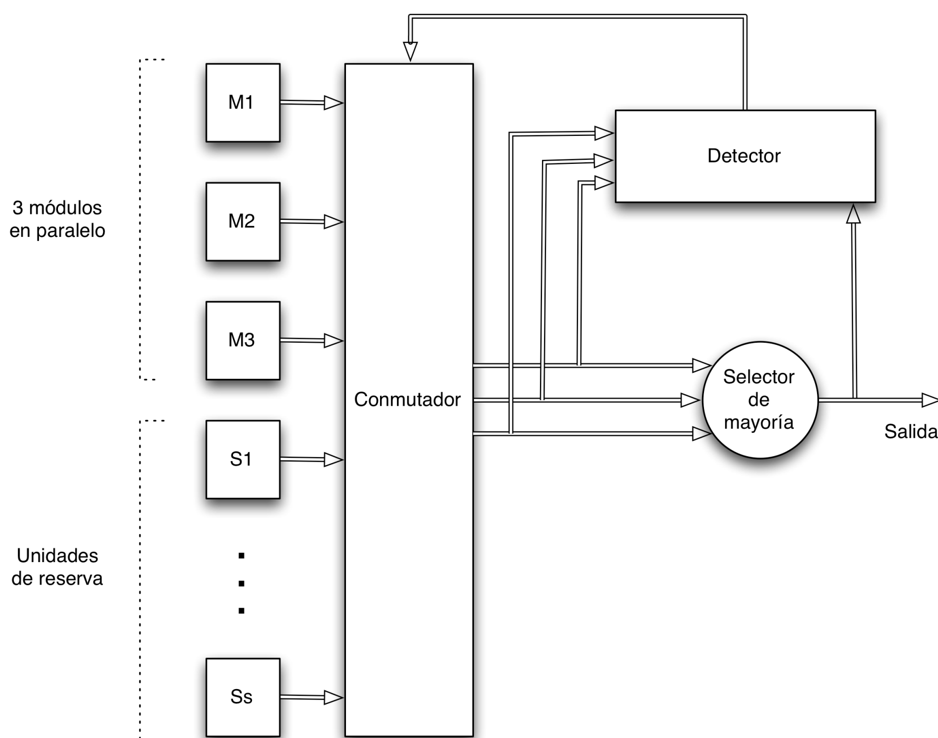


figura 9.15. Redundancia Híbrida (3,S).

9.9.3. Tolerancia a los fallos en el sistema lógico

Otra posibilidad para soportar tolerancia a fallos es recurrir al software. En este caso, el soporte lógico del sistema deben tener la capacidad de solucionar un fallo del equipo físico.

Existen dos ventajas fundamentales en la aplicación de este procedimiento. La primera consiste en que la tolerancia a fallos pueden introducirse en el sistema una vez diseñado el equipo físico. La segunda es la facilidad de modificar los requisitos de la tolerancia a fallos.

La principal desventaja de utilizar el software para tolerar los fallos es que no hay ninguna garantía de que el soporte lógico funcione correctamente cuando se produzca un fallo en el equipo físico, por lo que dependiendo de la naturaleza del fallo del equipo físico, dicho fallo podría ser irrecuperable, dejando bloqueada la máquina. Además, el coste del soporte lógico representa, en la actualidad, un porcentaje mucho mayor que el del equipo físico.

El método más característico de la tolerancia a fallos por software es el denominado "Vuelta atrás y reconfiguración" (*Rollback and Recovery*), que básicamente consiste en que si se detecta un fallo después de que haya sido ejecutada una instrucción, ésta se vuelve a ejecutar.

Como ya se ha comentado, al utilizar la tolerancia a fallos mediante software, un fallo en el equipo físico puede resultar desastroso para el sistema. De ahí que sea conveniente usar este tipo de tolerancia a fallos junto a la redundancia hardware para obtener una mayor fiabilidad del sistema.